Embedded Trace Macrocell

(Rev 0/0a)

Specification

ARM

ARM IHI 0014C

Embedded Trace Macrocell Specification

Release information

Change history

Date	Issue	Change
30th March 1999	A	First release
12th July 1999	B	Second release incorporating errata 01 corrections.
3rd December 1999	C	Third release incorporating protocol enhancements and modified trace port connector pinout.

Proprietary notice

Document confidentiality status

This document is Open Access. This document has no restriction on distribution.

Product status

The information in this document is Final (information on a developed product).

ARM web address

`http://www.arm.com`

Preface

This preface introduces the Embedded Trace Macrocell and its reference documentation. It contains the following sections:

About this document

This document is the Embedded Trace Macrocell Specification.

Intended audience

This document has been written for experienced hardware and software engineers who have some previous knowledge of the ARM architecture.

Organization

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the Embedded Trace Macrocell.

Chapter 2 *Trace Port Description*

Read this chapter for a description of the trace port protocol.

Chapter 3 *Triggering Facilities*

Read this chapter for a description of the triggering facilities that are available, and the different configurations that may be selected.

Chapter 4 *Programmer's Model*

Read this chapter for the JTAG register allocations and the programmer's model.

Chapter 5 *Trace Port Physical Interface*

Read this chapter for a description of the physical interface for the trace port, including signal timings and connector requirements

Typographical conventions

The following typographical conventions are used in this document:

bold	Highlights ARM processor signal names within text, and interface elements such as menu names. May also be used for emphasis in descriptive lists where appropriate.
italic	Highlights special terminology, cross-references and citations.
`typewriter`	Denotes text that may be entered at the keyboard, such as commands, file names and program names, and source code.
`typewriter`	Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands or functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications by ARM Limited.

ARM publications

ARM7TDMI Data Sheet (ARM DDI 0029)

ARM7TDMI-S Data Sheet (ARM DDI 0084)

ARM9TDMI Technical Reference Manual (ARM DDI 0145)

ARM Architectural Reference Manual (ARM DUI 0100)

Embedded Trace Macrocell (ETM9) Technical Reference Manual (ARM DDI 0157).

Other publications

Trace Port Analysis for ARM ETM User's Guide (HP Publications - publication number E5903-97000).

Feedback

ARM Limited welcomes feedback both on the Embedded Trace Macrocell, and on the documentation.

Feedback on this document

If you have any comments on this document, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Feedback on the Embedded Trace Macrocell

If you have any comments or suggestions about this product, please contact your supplier giving:

- the product name
- a concise explanation of your comments.

Contents
Embedded Trace Macrocell Specification

Chapter 1
Introduction

This chapter introduces the Embedded Trace Macrocell. It contains the following section:

- *Overview of the Embedded Trace Macrocell* on page 1-2.

1.1 Overview of the Embedded Trace Macrocell

This document describes the *Embedded Trace Macrocell* (ETM) for the ARM7 and ARM9 families of microprocessors.

The ETM consists of two parts:

A trace port A trace protocol has been developed to provide a real-time trace capability for processor cores that are deeply embedded in much larger ASIC designs. As the ASIC will typically include significant amounts of on-chip memory it is not possible to determine how the processor core is operating simply by observing the pins of the ASIC. A trace port is required to understand the operation of the processor.

Triggering facilities An extensible specification exists which allows users to specify the exact set of trigger resources required for a particular application. Resources include address and data comparators, counter and sequencers.

A software debugger provides the user interface to the ETM. The debugger allows all of the facilities (trace port and so on) to be configured via a JTAG interface and, where a trace port is implemented, displays the trace information which has been captured in a format that the user can easily understand.

Figure 1-1 on page 1-3 shows how a trace port is used in a complete debug environment. The debugger uses a JTAG interface unit to configure the ETM. The JTAG interface is also used for other debugging functions, such as downloading code and single-stepping through the program.

The ETM is used to compress the trace information and export it through a narrow trace port. An external *Trace Port Analyzer* (TPA) is used to capture the trace information.

Once the trace has been captured the debugger is responsible for extracting the information from the trace port analyzer and decompressing it to provide a full disassembly (with symbols) of the code that was executed. The debugger can also link this back to the original high level source code, thus providing the user with a visualization of how the code was executed on the target system.

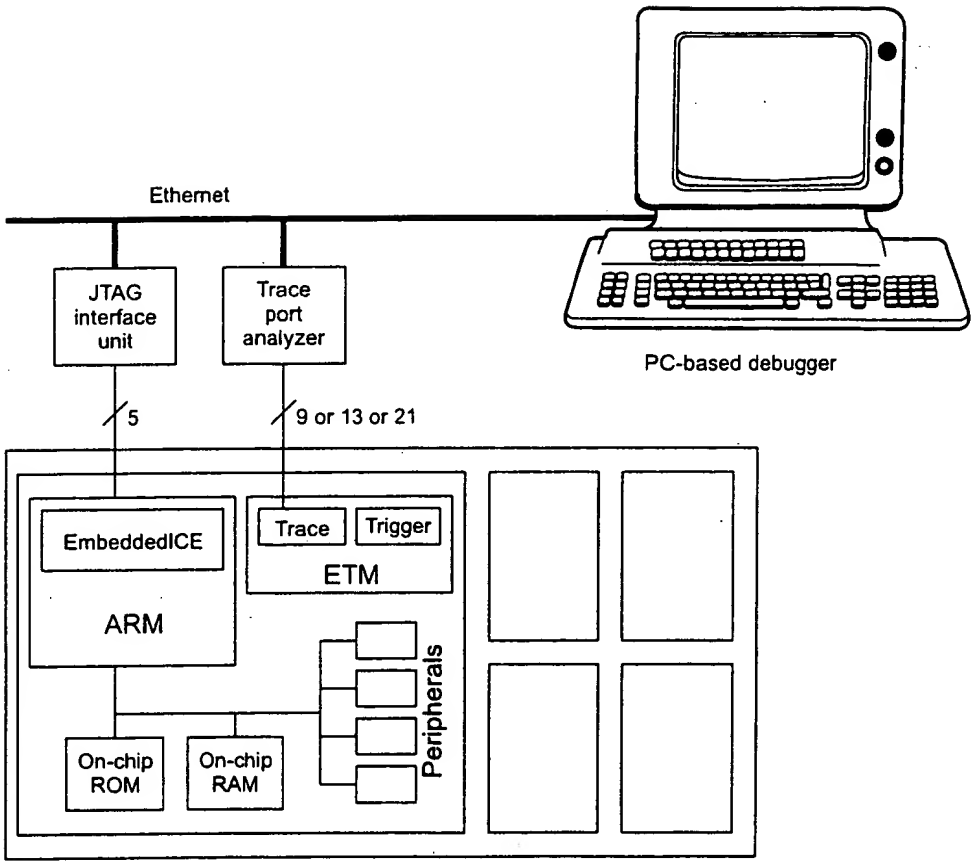

Figure 1-1 Example debugging environment

The debugging environment described uses a narrow trace port to reduce the number of additional pins that are needed on the ASIC.

The trace port compresses the trace information using the following techniques:

- Address information is only broadcast when the processor branches to a location that cannot be directly inferred from the source code.
- When an address is broadcast, high order bits that have not changed are not broadcast.

- Only data accesses that are indicated to be of interest are broadcast.
- For data accesses the user can select to broadcast only the data, only the address or both.

Note

For the debugger to be able to decode the trace, it must be provided with a static image of the code being executed. Self-modifying code cannot be traced because of this restriction.

1.1.1 Thumb support

Both ARM and Thumb instructions can be fully traced. The trace contains information about when the ARM core switches between ARM and Thumb state.

Chapter 2
Trace Port Description

This chapter describes the use of the trace port. It contains the following sections:

2.1 Introduction to the trace port

The trace port is used to broadcast two different types of trace information:

- instruction trace
- data trace.

2.1.1 Instruction trace

Instruction trace (or PC trace) shows the flow of execution of the processor and provides the user with a list of all the instructions that were executed. Instruction trace can be significantly compressed by only broadcasting branch addresses as well as a set of status signals that indicate the pipeline status (wait, execute, flush, and so on) on a cycle by cycle basis. Instruction trace is described in detail in *Instruction trace* on page 2-8.

2.1.2 Data trace

Data trace shows the data accesses performed by the processor, which occur as a result of the processor executing a load or store operation. For data accesses it is possible to broadcast both the address and the data. However, the user can choose to compress the data trace by only broadcasting either the address or the data. Data trace is described in detail in *Data trace* on page 2-13.

2.2 Structure of the trace port

Figure 2-1 shows the structure of the trace macrocell.

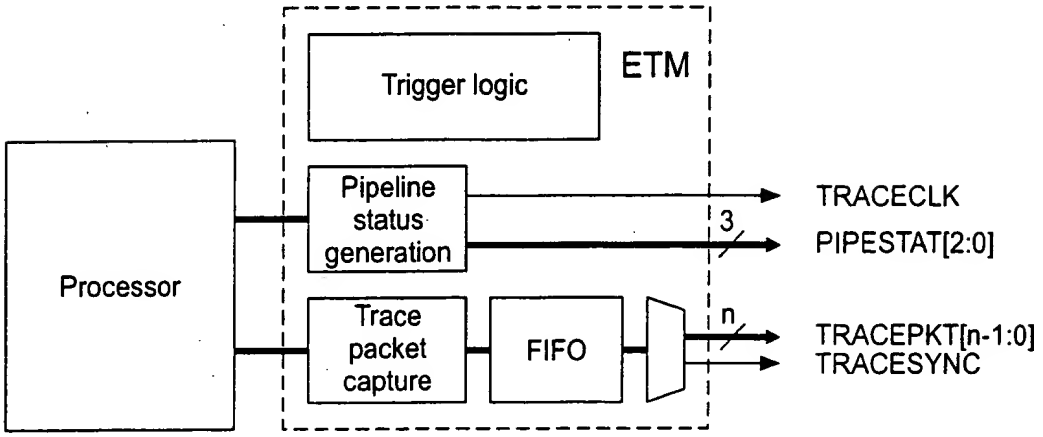

Figure 2-1 Embedded Trace Macrocell structure

The trace port comprises:

PIPESTAT three pipeline status pins

TRACEPKT an n-pin trace packet port, where n may be 4, 8 or 16 pins

TRACESYNC a trace packet synchronization pin.

These pins are referenced to the rising edges of the trace clock:

TRACECLK which is the same frequency as the processor clock.

The pipeline status signals provide a cycle-by-cycle indication of what is happening in the execute stage of the processor pipeline. The n-pin trace packet port provides additional information associated with particular pipeline status events. For example, if a change in instruction flow occurs then it is necessary to broadcast the destination address through the n-pin trace packet port. If the processor has executed an instruction which has failed its condition codes (all ARM instructions are conditional), no additional data is required through the trace packet port.

Separating the cycle-accurate pipeline status from the trace packets allows an on-chip FIFO to be used for the trace packet information. The FIFO can be used to buffer trace packets, for example when a number of branches occur in quick succession. Buffered packets can be passed out through the port when the processor is executing instructions

which have no trace packets associated with them, such as when a number of sequential instructions occur, in the following cycles. This technique allows the use of a trace port that has a lower data bandwidth than the maximum peak bandwidth.

The width of the trace packet port will be determined by the bandwidth of data trace that is required. A 4-pin port can be used when the number of data accesses to be traced is relatively low. The 8-pin and 16-pin variations of the port are more suitable when medium or high numbers of data accesses must be traced to provide the required debugging capabilities.

Use of the on-chip FIFO means that a particular pipeline status event and its associated trace packet (or packets) may not appear in the same cycle. Therefore, a synchronization signal is provided to indicate the start of certain sequences of trace packets and occasionally an offset is broadcast to allow synchronization of the two streams of information. This is described in more detail in the following sections.

2.3 Introduction to pipeline status

The pipeline status can be considered to be a view of the execute stage of the pipeline. Any side effects of the instruction, for example an aborted load or a write to the PC, are observed immediately, before the pipeline status for the next instruction is generated. This makes the protocol independent of the exact pipeline implementation of the ARM.

Each executed instruction generates a single pipeline status message on the **PIPESTAT** signals of the port. These are described in Table 2-1. A more detailed treatment of **PIPESTAT** messages is provided in *Pipeline status* on page 2-31:

Table 2-1 PIPESTAT messages

Status	Mnemonic	Meaning	Description
000	IE	Instruction executed	Usually this is used for an operation that did not cause a branch, but it is also used for direct branches. That is, where the destination can be worked out by referring back to the code image.
001	ID	Instruction with data	An instruction has been executed which also caused some data to be traced.
010	IN	Instruction not executed	An instruction has been executed but had no side effects because it failed its condition code test.
011	WT	Wait	Nothing useful happened in the cycle. This may be for a number of reasons, including memory system wait cycles and multi-cycle instructions.
100	BE	Branch executed	This is generated when an indirect branch is executed and therefore a destination address for the branch is required.

Table 2-1 PIPESTAT messages (continued)

Status	Mnemonic	Meaning	Description
101	BD	Branch with data	An instruction has been executed which caused a branch, that is the PC has been changed, and also caused some data to be traced.
110	TR	Trigger	Indicates a trigger condition has occurred.
111	TD	Trace disabled	Indicates that the trace can be disabled.

Instructions that are fetched but not executed, due to an executed branch or exception are not indicated as *instruction not executed*, but as *wait*. Only instructions which reach the execute stage of the pipeline are traced.

For multi-cycle instructions or for cycles on which the memory system has asserted a wait signal, *wait* cycles are indicated. See *Trace packets* on page 2-7 for a detailed description of the position and generation of packets with respect to the associated pipeline status.

The trigger and trace disabled status information types are described in more detail in *Trigger* on page 2-26 and *Pipeline status* on page 2-31.

The following sections go on to describe the exact behavior of the pipeline status and the way trace packets are generated on the **TRACEPKT** pins.

2.4 Trace packets

The **TRACEPKT** pins are used to output packaged address and data information related to the pipeline status. All packets are eight bits in length, irrespective of the number of **TRACEPKT** pins implemented. There are three cases to consider for how trace packets are output on the **TRACEPKT** pins:

Four pins: a packet is output over two cycles. In the first cycle packet [3:0] is output and in the second cycle packet [7:4] is output.

Eight pins: a packet is output in a single cycle on **TRACEPKT[7:0]**.

Sixteen pins: up to two packets can be output per cycle. If there is only one valid packet, it is output on **TRACEPKT[7:0]** and **TRACEPKT[15:8]** is unpredictable. If there are two packets to output the first is output on **TRACEPKT[7:0]** and the second on **TRACEPKT[15:8]**.

Additional pins are used to indicate the type of packet being output. **TRACESYNC** is HIGH on the first cycle of a PC address packet sequence. See *Instruction trace* on page 2-8 for more details. The rules used when outputting packets should also be considered. These are described in detail in *Trace packet generation* on page 2-21.

2.5 Instruction trace

Instruction trace works by broadcasting the destination address of branches. This information can be used along with the pipeline status signals to determine how many instructions after each branch are executed.

Instruction trace is described under the following:

- *Direct and indirect branches*
- *Branch reason code* on page 2-10
- *Address packet offset* on page 2-10
- *Full address broadcast* on page 2-11
- *Exceptions* on page 2-12.

2.5.1 Direct and indirect branches

For certain branches it is not necessary to broadcast the destination address. For direct branches (`B` or `BL` instructions) the assembler code provides an offset to be added to the current PC. It is only necessary to know the address of the instruction along with the fact that it executed and the destination of the branch can be calculated.

The branch address only needs to be broadcast when the program flow changes for some reason other than a direct branch. These are collectively known as indirect branches. Examples of this are:

- a load instruction (`LDR` or `LDM`) with the PC as one of the destination registers
- any data operation (`MOV`, `ADD`, and so on) with the PC as the destination register
- a `BX` instruction, which moves a register into the PC
- a `SWI` instruction or undefined instruction
- all other exceptions, such as interrupt, abort or reset.

Exceptions and mode changes (between ARM and Thumb) are understood by broadcasting the destination address. No attempt is made to specifically indicate the exception entry since this cannot always be determined from the external ARM signals.

Bit 0 of the address is used to show whether the destination of the branch is ARM code (bit 0 LOW) or Thumb code (bit 0 HIGH).

Bit 1 of the address will be traced as zero for ARM instruction accesses, regardless of the alignment of the instruction address. The decompressor should ignore bit 1 of the address for ARM instruction accesses. This bit of the address is reserved for future expansion.

The ETM monitors the processor memory interfaces to track the internal pipeline. This allows it to detect a nonsequential transfer in order to identify when the flow of execution is changed. It also performs a small amount of instruction decode for every

instruction that is fetched to detect B and BL instructions. Whenever the ARM changes its flow of control it is assumed to be a direct branch if the instruction decode showed a B or BL instruction and the destination of the branch is not in either of the vector tables. Otherwise the branch is considered indirect and a branch address is broadcast through the trace port.

Only an indirect branch requires a number of trace packets to provide the destination address.

A direct branch to an address in the vector table is traced as an indirect branch.

Compressed branch address packet structure

When a processor performs a branch operation the destination of the branch is often reasonably close to the current address. The spatial locality of branch destinations is used to provide further compression of the branch addresses. It is only necessary to broadcast the low order bits that have changed since the last branch and the full address can be reconstructed when decompression of the trace information takes place.

All trace packets are 8-bits in length and a branch address can be made up of between one and five packets. The **TRACESYNC** signal is used to indicate the first packet and is asserted HIGH only for the first packet of any branch address.

Each packet of a branch address is structured such that the most significant bit (bit 7) indicates if there are further address packets and this makes it possible to detect the last packet. If bit 7 is HIGH then a further address packet will follow. Bit 7 LOW means it is the last address packet.

To decide how many packets are required the on-chip logic registers the last branch address that it has broadcast and, when another branch occurs, the new address is compared with the one that was previously broadcast. Only sufficient low order bits need to be broadcast to cover all the bits that have changed in the address. For example, if the upper 12 bits of the address are unchanged and A[19] is the most significant bit to have changed then it is only necessary to broadcast A[19:0], which can be done in three address packets instead of five.

A full 32-bit address is made up of five packets, the first four will have bit 7 HIGH and the last packet will have bit 7 LOW. The address is made up as follows:

Address[6:0]	= bits 6 to 0 of first packet	bit 7 HIGH
Address[13:7]	= bits 6 to 0 of second packet	bit 7 HIGH
Address[20:14]	= bits 6 to 0 of third packet	bit 7 HIGH
Address[27:21]	= bits 6 to 0 of fourth packet	bit 7 HIGH
Address[31:28]	= bits 3 to 0 of last packet	bit 7 LOW

Note

When an address is broadcast which is less than 32-bits the new address value is used to replace the appropriate bits in the previously broadcast branch address. The value does *not* need to be added to or subtracted from the previous value, nor is it based on the immediately preceding PC value.

2.5.2 Branch reason code

Bits 6 to 4 of the fifth packet are used to provide a reason code. This indicates why the full branch address has been generated.

Table 2-2 Branch reason codes

Bits 6 to 4	Description
000	A normal PC change
001	Tracing has been enabled
010	Trace restarted after a FIFO overflow
011	The ARM has exited from debug state
100 - 111	Reserved for future expansion

2.5.3 Address packet offset

Every time the processor performs a branch operation at least two instructions that have been fetched into the pipeline will be discarded. As discussed earlier, these instructions are always indicated as *wait*, rather than *instruction not executed*. In these two cycles the **PIPESTAT** signals are reused to broadcast an *address packet offset* (APO). Wait states may result in more than two *wait* cycles following a branch. No attempt is made to use these additional cycles to output a larger offset.

The address packet offset information is used when decompressing the trace information to synchronize between the pipeline status (**PIPESTAT**) and the trace packets (**TRACEPKT**).

When decoding the trace the decompressor uses the branch offset to skip the specified number of addresses, including the cycle on which the branch pipeline status is generated. An offset of zero specifies that the next **TRACESYNC** in the trace determines the first packet of the address. An offset of one indicates that the second **TRACESYNC** in the trace identifies the start of the associated branch packets, and so on.

To ensure that the pipeline status can be successfully decompressed unambiguously it is necessary to only use certain values on the **PIPESTAT** signals when broadcasting the APO. Therefore, only an offset of between 0 and 3 is used on each cycle. Table 2-3 on page 2-31 and Table 2-4 on page 2-32 define the pipeline status types and their bit encoding. The IE, ID, IN and WT types are reused. BE, BD, TR and TD must not be reused. This is because they may be issued at any time.

If a BE is observed in an address packet offset cycle this indicates that the previous BE has been abandoned. The address for this abandoned branch will still be output, and should be ignored.

A TD will not be observed in an address packet offset cycle. A TR may be issued at any time and therefore the APO will be broadcast on the **TRACEPKT** pins, as defined in *Trigger* on page 2-26.

The two address packet offset cycles allow an offset between 0 and 14, with the value of 15 being reserved to indicate that the offset is too large to be encoded. During the first cycle after a branch the least significant two bits of the offset are broadcast and the most significant two bits are broadcast during the second cycle.

2.5.4 Full address broadcast

For large trace captures it is likely that the first trace sample will have been lost from the trace port analyzer (overwritten by a newer trace sample) by the time that the trigger event occurs. For this reason it is necessary to occasionally broadcast a full 32-bit address in order for the decompression of the trace to be accomplished successfully.

A cycle counter of no more than 1024 must be implemented. When this counter reaches its maximum count of 1024 a full address is broadcast, for an indirect branch, at the next opportunity, but only if the FIFO is empty enough to accept the five trace packets required without overflowing. If this is not possible then the branch is treated normally. This process continues for up to 512 cycles until there is enough space in the FIFO to accept a full 32-bit address. If a full address has been generated the counter is reset.

If there were no indirect branches (such as in a large program loop), or the FIFO has not emptied sufficiently, then a full address will not have been generated. In this case a full five-packet address is forced by either:

- Turning the next direct branch into a five-packet indirect branch. The direct branch is made indirect by generating a BE pipeline status.
- Forcing the next indirect branch to be a five-packet branch, even if this was not scheduled.

Both of these measures could cause an overflow to occur. However, this drawback is a reasonable penalty for ensuring synchronization.

2.5.5 Exceptions

No explicit information is generated in the trace to indicate that an exception has been taken and the probable corresponding mode change that goes with it. The instruction that is cancelled or affected by the exception becomes a branch to the exception vector. *Exception behavior* on page 2-20 describes the exact behavior for each type of exception.

2.6 Data trace

Data trace cannot use all of the compression techniques that are available with instruction trace and tracing every data access could require a large number of pins to achieve the required bandwidth. There are three ways in which data trace can be optimized. This is accomplished using the following techniques:

- *Data access filtering*
- *Address/data selection*
- *Address compression* on page 2-14.

2.6.1 Data access filtering

The main technique that is used to reduce data trace is for the user to select which data accesses need to be observed. The triggering facilities generate an internal function/ signal called **ViewData** to indicate that the data from individual locations or address regions are traced. See *Data tracing* on page 3-16 for details of the generation of **ViewData**.

For load and store multiple operations (`LDM` and `STM` instructions) **ViewData** is only sampled for the first access of the sequence. This ensures that either none or all of the words transferred are traced. This is necessary for successful decompression, because the transferred data has to be associated with the correct ARM registers. This also applies to coprocessor load/store (`LDC`/`STC`) instructions. It is important that the number of words transferred for an `LDC` or `STC` instruction can be statically determined from the instruction. The debugger must be aware of this mapping.

Data filtering is not always precise and it is sometimes not possible to trace the event that is used to control data filters, see *Trace filtering* on page 3-15.

2.6.2 Address/data selection

The second technique that can be used to reduce the bandwidth of the data trace is to choose what information is broadcast for each data access. The following three options are available:

- *Address only*
- *Data only* on page 2-14
- *Address and data* on page 2-14.

Address only

The address only option just broadcasts the address of the transfer, or the first address in the case of a load/store multiple. This approach is useful when checking the code to ensure that the correct address is generated for all transfers.

Data only

The data only option just broadcasts the data of the transfer. The amount of data broadcast will be appropriate for the instruction executed:

- A load/store byte would require just one 8-bit data packet.
- A half-word would require 2 packets.
- A word would require 4 packets.
- A load/store multiple would require n * 4 packets, where n is the number of registers transferred.

This approach is useful when there is a high degree of confidence that the address of the transfers is being generated correctly and the address of a transfer can easily be inferred by looking at the instruction that caused the access. For example if the instruction was in the middle of a section of code for the UART mode control it can be assumed that the address is that of the UART control register.

Address and data

The address and data option broadcasts both the address and data of the transfer. This ensures that all information about the transfer is known, however this does require a larger overall bandwidth through the trace port.

Because of the techniques that are used to decompress the trace information after it has been captured, the above modes of operation must be selected for all data transfers. For example, in a single trace it is not possible to provide the address only for some transfers and data only for others.

2.6.3 Address compression

The final technique that can be used to compress the data trace is to reduce the number of bits that are broadcast for the address of the data transfer. The same technique is used as for branch addresses, whereby a copy of the last data access address is kept and only the low order bits that have changed are broadcast for the next address.

This technique is particularly effective when data is being viewed in just one relatively small address range as all the traced data accesses will have the same high order address bits.

When the address of a data access is about to be broadcast it will only be compressed if a full 32-bit data address has been broadcast in the last 1024 cycles, and there has been no interruption in tracing. Otherwise no compression will take place and the full 32-bit address will be used to ensure that the captured trace information will contain a reference point for the other compressed address packets.

2.7 Data trace considerations

The following should be considered when using data trace:

- *Data accesses in the pipeline status signals*
- *Data trace packets* on page 2-17
- *Data aborts* on page 2-17
- *Decoding the data trace packets* on page 2-18
- *Tracing of unaligned loads* on page 2-18.

2.7.1 Data accesses in the pipeline status signals

In order to work out which load and store instructions caused packets to be inserted into the trace stream specific pipeline status encodings are generated.

A load/store operation would be signalled in four possible ways, depending on the state of **ViewData** and whether the instruction will cause a branch. These are:

- *Instruction executed*
- *Instruction executed with data*
- *Branch executed with data*
- *Branch executed.*

Instruction executed

This indicates that an instruction executed and either the instruction was not a load/store operation or, if it was, the data access was not placed into the trace stream as **ViewData** was inactive.

Instruction executed with data

This indicates a load/store instruction has executed causing a data access and **ViewData** was active when the access occurred.

Branch executed with data

This special case is similar to the instruction executed with data, except that the instruction also caused a write to the PC. This may be due to an explicit load operation to the PC, or that the load/store was aborted. Both of these occurrences will cause the processor to branch and therefore the trace packets will need to include a branch destination address as well as any packets associated with the data transfer.

Branch executed

This indicates that a load to the PC has been executed without **ViewData** being active when the access occurred.

2.7.2 Data trace packets

A data access can cause a number of different types of trace packet to be inserted into the trace packet stream. If the trace port is configured to capture the address of data accesses then this is the first information to be placed into the trace stream. Between one and five trace packets are required to encode the address, in a similar manner to branch addresses, and bit 7 of each packet indicates if another address packet is to follow.

Note

TRACESYNC is *not* asserted when the load/store address is output.

The second piece of information to be placed in the trace stream is the actual data that is used in the transfer. As mentioned above, the number of 8-bit data packets will be the same as the number of bytes transferred by the instruction. Again this information is only broadcast if the trace port is configured to provide address and data or data only.

When the instruction is a load operation with the PC as a destination register a branch destination address is also required. This is encoded in the same fashion as all other branch addresses and will be the last of the three types of trace packet to be broadcast. **TRACESYNC** will be asserted when the first of the instruction address packets is output, in the same way as for other branches.

Note

The swap and swap-byte instructions, `SWP` and `SWPB`, sample **ViewData** at the beginning of the instruction. As a result either both the word or byte transfers, or neither, will be traced.

2.7.3 Data aborts

If one or more of the data accesses was aborted by the memory system, a PC address will also be broadcast as part of the same instruction. See *Full address broadcast* on page 2-11 for details.

A data abort may occur on any or all of the data transferred. Data tracing ignores the abort status of words transferred. All transferred words of an instruction, whether aborted or not, are traced.

The pipeline status for the aborted instruction will be *branch executed* or *branch with data* depending on whether there is any traced data associated with the instruction.

2.7.4 Decoding the data trace packets

When decompressing the captured trace information one of the first tasks that must be carried out is to realign all the pipeline status events with the various trace packets that have been broadcast. In order to do this the decompression tool must find a synchronization point and then work through the trace information working out which packets are associated with which events.

The decompression tool must know in advance whether a data access will have address packets, data packets or both. The tool can work out how many address packets are needed by looking at bit 7 of each packet. To determine how many data packets are required it is necessary to calculate the address of the instruction and then decode the instruction from the code image. Once the instruction is known it is possible to work out how many data packets are required. In the case of a load/store multiple it will be necessary to determine how many registers are being transferred to work out how many data packets are expected.

2.7.5 Tracing of unaligned loads

The ARM will, under some circumstances rotate the incoming data. It does this based on:

- the current endian configuration
- the type of instruction
- the size of the transfer
- bits 0 and 1 of the address from which the data is loaded.

In addition the ARM selects and rotates different parts of the data bus for half-word and byte accesses, described in *Endian effects* on page 2-28. For clarity some specific points on load instructions are worth noting:

- Word LDR instructions rotate the incoming data, based on bits 1 and 0 of the address bus. The ETM will not rotate the data. The data is traced as it is provided to the ARM by the memory system, not as it is loaded into the ARM register. This is to remove the need for the ETM to decode the ARM instruction type.
- Halfword LDR instructions select the addressed halfword and are unpredictable if the address is not halfword aligned. The ETM will trace the appropriate halfword, ignoring bit 0 of the address.
- Byte LDR instructions select the addressed byte. The ETM will trace the addressed byte.

- LDM instructions do not rotate the incoming data if the address is not word aligned. The ETM will trace the word returned by the memory system.

2.8 Exception behavior

If the cancelled instruction is a branch the trace decompressor must assume it to be caused by the appropriate exception if the address resides within the vector table. For processors that support vector table relocation then the vector table resides at 0x00 to 0x1c if **HIVECS** is LOW, or 0xffff0000 to 0xffff001c if **HIVECS** is HIGH.

The possible exception types and the corresponding trace behavior are:

Reset — This will cause the current instruction to be abandoned. When reset is de-asserted a branch occurs to the reset vector.

Interrupts (IRQ and FIQ) — These will cause the interrupted instruction to become a branch to the appropriate exception vector.

Prefetch abort — When the aborted instruction is executed, it becomes a branch to the abort vector.

Data abort — The instruction on which the data abort is signalled becomes a branch to the data abort vector. The pipeline status for the instruction will be a BE or BD, depending on whether any data was traced.

Undefined instructions — When the undefined instruction is executed, it becomes a branch to the undefined instruction vector.

SWI — When the SWI instruction is executed, it becomes a branch to the SWI vector.

Note

Whenever any of the above are traced, a BE **PIPESTAT** is generated and the next instruction traced will be the instruction at the exception vector.

For some ARM processors, tracking the pipeline is not always possible under some exception sequences. This may result in an exception being traced by changing the pipeline status for the last instruction executed to a BE/BD to the exception vector. Consult the specific ETM Technical Reference Manual for details.

2.9 Trace packet generation

This section describes the restrictions and requirements for generating the trace packets. This area of the specification is very important because the software that decompresses the trace must always be able to work out when there is valid data on the **TRACEPKT** pins, and which instructions these apply to. In particular the decompressor needs to determine whether there is valid **TRACEPKT** data on cycles with a pipeline status other than *wait*.

The rules are:

- Packets generated by a particular instruction must form a continuous block in the packet stream.
- Gaps in this block are only permissible when the **PIPESTAT** for a particular cycle would normally be WT, and in this case the **PIPESTAT** is changed to be TD, indicating that there is no data in the trace packet. Hence, when TDs are filtered out, the block of packets appears continuous. This is not the case when cycle-accurate tracing is enabled. See *Cycle-accurate tracing* on page 2-30.
- The group of packets for a particular instruction may not start on or before any **PIPESTAT**s generated by a previous instruction. This includes any APO cycles for that instruction (for example a BE followed by an ID would cause the packets to be delayed until after the BE and APOs). If an instruction generates *wait* **PIPESTAT**s before generating its functional **PIPESTAT** (ID, BE or BD), the packets for this instruction may begin on any of these *wait* cycles (observing also rule 2, to ensure that any gaps in the packet stream are indicated).
- When the FIFO is not empty, packets generated by a particular instruction must be generated immediately after any data already in the FIFO is output. That is, there must be no gaps between the packets.
- When the FIFO is empty the start of the group of packets for a particular instruction must occur no later than the associated **PIPESTAT**.

Note

If a trigger occurs, all subsequent trace packets will be delayed by one cycle. This cycle is used to output the trigger response.

2.9.1 Additional considerations for 16-bit ports

A 16-bit port has to be treated slightly differently to deal with the possibility of port transactions that do not use the full port width.

The following rules apply:

- A single packet is only output if the **PIPESTAT** is not *wait*.
- The first packet of a branch address must always be broadcast on **PIPESTAT[7:0]**.

There are three exceptions to these rules, when it can be guaranteed that the next trace packet will have an associated **TRACESYNC**:

- When the FIFO is being drained after an overflow has occurred. The decompressor needs to be aware that FIFO draining may generate an empty byte. The address packet offset for this BE may not be zero if the FIFO has not drained when tracing is enabled.
- The ARM is in debug state. This is identified by **DBGACK** being HIGH.
- When the FIFO is being drained following tracing being disabled.

2.10 Enabling and disabling trace

The ETM has a **TraceEnable** function that allows tracing to be enabled or disabled. This signal would typically be used to select the areas of code that are traced and disable the trace when code is executed which is of limited use to the debugging process. The advantage of disabling the trace information is that it effectively increases the amount of useful information that can be captured by a given size of buffer in the trace port analyzer, thus allowing selective tracing over a longer time period.

2.10.1 Enabling trace

When **TraceEnable** becomes active tracing is enabled. The trace port will broadcast a *branch executed* status and associated with it will be a full 32-bit address packet. This provides a start address so that the trace information can be successfully decompressed from the first instruction after the trace is enabled. This indicates to the trace decompressor software that there is a discontinuity in the trace. A correct address packet offset must be generated, because the FIFO may not be empty at the point at which tracing has been enabled.

When tracing starts the instruction address generated is identified by one of two reason codes:

- tracing has been enabled
  The address packet offset for this BE may not be zero if the FIFO has not been drained when tracing is enabled.
- trace restarted after a FIFO overflow

  The address packet offset for this BE will be 0, and **TRACESYNC** is asserted in the same cycle that **PIPESTAT** = BE.

The first BE to be generated is not a real branch but is used as a speculative entry point. The instruction address may be proved correct, in which case the target instruction has successfully executed and is traced. If however the instruction is not executed a *real* branch (direct or indirect) has occurred. At this point the trace port abandons the speculative branch, *even if it is in the middle of broadcasting the address packet offset*, and outputs a **PIPESTAT** of BE, and the new full 32-bit address is sent to the FIFO, with the same reason code. For this branch, the address packet offset may be 1 or 0.

2.10.2 Disabling trace

When **TraceEnable** becomes inactive tracing stops and the FIFO status changes to *wait* whilst the FIFO drains. When there is no further data in the FIFO the pipeline status changes to *trace disabled*. This allows the trace port analyzer to suppress tracing, and so improve the trace buffer utilization.

Note

In cycle-accurate tracing *trace disabled* cycles can correspond to *wait* cycles, in which case the trace port analyzer may still need to capture these cycles. For more information see *Cycle-accurate tracing* on page 2-30.

2.10.3 Data accesses during disabled trace

When the trace port is disabled it is not possible to view data accesses and the **ViewData** output will be ignored.

2.11 FIFO overflow

Under certain circumstances it is possible that so much trace information is generated on-chip that the FIFO can overflow. When this occurs a two-stage process to empty the FIFO and restart the trace takes place:

1. First the pipeline status is changed to *wait* and the FIFO is allowed to empty. This ensures that all trace information up to the overflow condition is collected, which may be useful in determining the cause of the FIFO overflow. If overflow occurs part way through a load or store instruction the pipeline status for the instruction must be generated. This is to allow the decompression software to associate any trace packets with an instruction.
2. When the FIFO has been allowed to drain, tracing should be re-enabled as soon as possible (assuming **TraceEnable** is still active).

The trace decompressor can successfully decode a FIFO overflow sequence, if it is aware that a BE marked as *FIFO overflow* is speculative and may be followed by another BE also marked as *FIFO overflow*. If this is the case, then the second BE marked as *FIFO overflow* may overwrite the address packet offset of the speculative BE. This is not a problem, since the speculative BE was found to be incorrect, and therefore can be discarded.

From version 1 of the protocol, an ETM status register is provided. Bit 0 of this register provides a pending overflow flag, indicating that an overflow has occurred but that an *overflow occurred* reason code has not been generated. This is needed where tracing stops due to an ARM breakpoint, but before tracing can be restarted.

2.11.1 System stalling

It is desirable to indicate to the system that the FIFO is about to overflow. An on-chip **FIFOFULL** output is provided which indicates when the FIFO has less than a configured number of bytes of space available. The assertion of this signal is controlled by configurable instruction address regions to prevent system stalling in critical code. See *FIFO overflow* on page 3-19 for details.

2.12 Trigger

A trigger point is used by the trace port analyzer to decide when to stop the trace capture process. If a trigger is not used it is possible that any interesting trace information could be overwritten by newer trace samples before the user is able to stop the tracing process.

The external trace port analyzer can use the trigger in several ways:

Start trigger — A start trigger will capture trace data from the trigger point onwards, thus capturing what happened after the trigger event.

Stop trigger — A stop trigger will stop the tracing, thus the trace port analyzer will be full of the events that happened before the trigger event.

Centre trigger — A trigger position between a start trigger and a stop trigger will capture a certain number of events before and a certain number of events after the trigger event.

The trigger condition is determined using the full range of triggering resources in the ETM, and is discussed in *Trigger resources* on page 3-5.

Trace discontinuities (overflow of the FIFO and **TraceEnable** being inactive) will result in the trigger condition not being precise. Decompression of the trace will not therefore associate the trigger with a particular processor cycle. In addition if an instruction causes the trigger to occur, it is not guaranteed that the trigger status will be generated on the pipeline status for that instruction.

2.12.1 Trigger packet

Rather than having a dedicated pin to indicate a trigger event, a special pipeline status encoding is used.

The trigger pipeline status (TR) replaces the current pipeline status. The pipeline status that is replaced is broadcast on the **TRACEPKT[2:0]** pins. The FIFO draining is stopped for that cycle to allow this to happen. The decompressor must take account of this.

For a trigger pipeline status, when a *wait* would have been generated the *wait* becomes *trace disbled*. That is, WT + TR $\Rightarrow$ TR + TD.

2.13 Coprocessor operations

There are three different types of coprocessor instruction that must be considered:

- coprocessor data operation (CPDO)
- coprocessor data transfer (CPDT)
- coprocessor register transfer (CPRT).

2.13.1 Coprocessor data operation (CPDO)

These operations occur when the processor executes a CDP instruction. A CPDO instructs a coprocessor to perform an internal data operation. This cannot produce any trace packets and is treated exactly the same as any other instruction.

2.13.2 Coprocessor data transfer (CPDT)

These transfers occur when the processor executes an LDC or STC instruction. CPDT instructions are treated in the same way as standard processor loads and stores. Only when looking at the disassembled trace information will it be apparent that the data access involved a coprocessor.

When decompressing the trace stream it is necessary to know how many data packets are associated with a load or store. In the case of a coprocessor load/store it is necessary to have specific knowledge of the coprocessor involved because the number of data packets is not directly encoded in the instruction and is determined by the coprocessor during execution of the instruction.

2.13.3 Coprocessor register transfer (CPRT)

These transfers occur when the processor executes an MCR or MRC instruction and are used to move data between the processor registers and the coprocessor. There is no address related to a CPRT and the data size is always 32-bits. As there is no address, **ViewData** cannot be used to determine if these transfers should be placed in the trace stream and is not considered. Instead, a configuration bit is provided to select whether or not the data should be captured. If the data is traced the pipeline status is ID. If no data is traced it is IE.

2.14 Endian effects

The trace port must take account of the endian configuration of the ARM core and the address and type of memory requests. The ARM samples different parts of the data bus, based on bits 1 and 0 of the address of the transfer, the current endianness and the size of the transfer.

Some systems comprise a mixture of big and little-endian memory and peripherals. The endian configuration may therefore change during execution (under the control of the ARM) depending on the general memory region being accessed. As a result, endianness must be considered separately for each transfer.

Data is inserted into and read out of the FIFO least significant byte first. Data is presented on the trace port in the order it is read out, so for a 16-bit wide port, the first byte is presented on port [7:0], and the second byte on port [15:8]. For a 4-bit wide port, bits three to zero are output first, followed by bits seven to four.

If these rules are observed, then for the data, the trace decompression software does not need to know about the endianness of the system, which could change dynamically based on the peripherals being accessed. However, for the instructions, when decompressing the trace, the endianness of the instruction code regions must be known.

2.15 Debug state

When the ARM enters debug state instruction execution stops. This means that tracing must also stop. The FIFO will continue to drain until empty.

In debug state, system speed accesses cannot be traced and so are ignored by the trace port.

When the ARM exits debug state, if tracing is enabled, tracing will restart with reason code 011, *The ARM has exited from debug state.*

If an overflow has occurred on entry into debug state, then the debug tools can detect this by reading the ETM status register, for details see *ETM status* on page 4-9 and *FIFO overflow* on page 2-25.

Refer to the specific ARM datasheet or technical reference manual for details of debug state.

When in debug state, TD cycles will always have **TRACEPKT[0]** LOW to prevent TD cycles from being captured.

2.16 Cycle-accurate tracing

When profiling the execution of critical code sequences it is often useful to be able to observe the exact number of cycles that a particular code sequence takes to execute. To be able to do this, the trace port analyzer must know that it must not suppress the capturing of *trace disabled* (TD) cycles. This is indicated by **TRACEPKT[0]** being HIGH during TD cycles.

During periods where tracing is disabled (**TraceEnable** is inactive) TD cycles will always have **TRACEPKT[0]** LOW. For TD cycles, when **TraceEnable** is active, bit 12 of the ETM control register (see Table 4-2 on page 4-7) controls whether **TRACEPKT[0]** is HIGH or LOW. This allows the user to select whether, in the regions in which tracing is required, the cycle count of the program execution in the trace will be correct.

When in debug state cycle-accurate tracing is disabled.

2.17 Pipeline status

This section fully defines how the pipeline status signals are used. Table 2-3 describes all the pipeline status options.

Note

For the two cycles after a branch (BE or BD) the pipeline status pins give an address packet offset indicating how may branch addresses are currently in the on-chip FIFO.

Table 2-3 Pipeline status encoding

Status	Mnemonic	Description
000	IE	Instruction executed. Indicates an instruction has been executed which has not generated any associated trace packets.
001	ID	Instruction executed with data. Indicates a load or store instruction has executed, where the address of the data access has been flagged so that the access is broadcast on the trace port.
010	IN	Instruction not executed. This is used to indicate that an instruction has reached the execute stage of the pipeline, but has failed its condition codes.
011	WT	Wait. The pipeline has not advanced either because the memory system has asserted the wait signal to the processor, or because the processor is performing an internal cycle. Wait cycles are also generated when tracing is disabled. If the on-chip FIFO is empty this status is replaced by *trace disabled.*
100	BE	Branch executed. Used whenever the processor branches to a location that cannot be directly inferred from the source code. The trace port may optionally be switched into a mode where it will broadcast all branches.
101	BD	Branch executed with data. Used whenever a data access causes a branch, which will occur when a load instruction has the PC as the destination register.
110	TR	Trigger. An on-chip trigger has occurred.
111	TD	Trace disabled. The trace may be disabled to prevent the trace port analyzer from filling up with unwanted information. This encoding is used when the trace is disabled or when there is no packet being broadcast.

Table 2-4 describes the various packets that can be associated with different types of pipeline status.

Table 2-4 Trace packet types

Status	Associated trace packets
IE Instruction executed	No packets.
ID Instruction executed with data	There are three modes of operation for tracing data accesses: Data only: 1 packet for bytes, 2 packets for halfwords, 4 packets for word transfers. N × 4 packets for load/store multiple operations, where N is the number of registers transferred. Address only: 1 to 5 packets containing the address of the transfer. For load/store multiples this will be the address of the first transfer in the sequence. Data and address: A number of packets for address followed by a number of packets for data, as described above.
IN Instruction not executed	No packets.
WT Wait	No packets.
BE Branch executed	1 to 5 packets containing the branch destination. The most significant bit of each packet, bit 7, indicates if an additional packet is to follow, When bit 7 is HIGH there is an additional packet, when bit 7 is LOW it is the last packet. For a five-packet branch destination bits 6:4 of the final packet are used to indicate the reason for the branch address, for example a trace discontinuity.
BD Branch executed with data	A number of packets providing the information on the data access, as for the *instruction executed with data* case described above, followed by 1 to 5 packets containing the branch destination.
TR Trigger	The **TRACEPKT[2:0]** pins are used to output the pipeline status that would have been generated.
TD Trace disabled.	No packets. **TRACEPKT[0]** is used to indicate the status of **TraceEnable** when cycle-accurate tracing is enabled.

Chapter 3
Triggering Facilities

This chapter describes the triggering facilities available with the Embedded Trace Macrocell. It contains the following sections:

3.1 Introduction to triggering

This chapter specifies the triggering in the ETM. The triggering resources are used to control the trace port, providing both triggering and filtering functionality to enhance the usefulness of the trace. This is often very important when a narrow trace port is used, because careful selection of the data and instructions to trace is required, if the trace port bandwidth is not to be exceeded.

The trigger module works by monitoring the address and data buses and the pipeline of the processor and generates a number of signals to control the trace port. These signals are used to:

- signal a trigger
- filter the instruction trace by enabling/disabling the trace port
- filter the data trace by indicating which data accesses need to be traced.

The trigger hardware, like the trace port, monitors the processor's pipeline to determine if or when a fetched instruction is executed. This is important because instructions prefetched after a branch, or the *random* occurrence of an interrupt can cause instructions to be fetched many times without being executed.

3.2 Trigger outputs

When the trigger event occurs, a trigger pipeline status is generated on the **PIPESTAT** signals as soon as possible. This indicates to the trace port analyzer that the required event has occurred.

A simple trigger might be based on memory access address or data matches, such as an instruction from a particular address is executed. However, a more complicated set of trigger conditions are allowed, such as a particular instruction being executed a number of times or a particular sequence of events occurring.

The trace port analyzer can use the trigger in the following ways:

- *Trace after*
- *Trace before*
- *Trace about*

3.2.1 Trace after

The trigger can indicate that the trace information should be collected from the trigger point onwards. This is often called a start trigger and would be used if you want to know what happens after a particular event, for example what happens after entering an interrupt service routine. A small amount of trace data is often also collected before the trigger condition.

3.2.2 Trace before

The trigger can be used to stop collection of the trace. In this instance the trace port analyzer acts like a large FIFO, so that it always contains the most recent trace information and the older information overflows out of the trace memory. In this case the trigger indicates that the FIFO should stop, so the memory contains all the trace information before the trigger event. This is often called a stop trigger and would be used to find out what caused a certain event. For example, to see what sequence of code was executed before entering an error handler routine. A small amount of trace data is often also collected after the trigger condition.

3.2.3 Trace about

The trigger can be set between the start point and the stop point, such that the trace memory contains some information before the trigger point and some information after it.

Only a single trigger can be generated in any trace run, once the trigger has been asserted it is necessary to reprogram the triggering before another run can begin.

3.2.4 Tracing on/off

For any particular trace run the user can select when the trace is enabled and disabled. This allows best use of the trace memory, as it is not used up when the trace port is disabled. For example, the trace port could be enabled only when a particular function is executing and disabled at all other times. If this function occurs infrequently and trace was always on it may only be possible to catch one or two occasions when the function executes in any particular trace session. However, by disabling trace when the function is not executing it is possible to ensure that the trace memory is only filled with relevant information and many more instances when the function is executed will be captured, extending over a greater period of time.

Whenever the trace is re-enabled the trace port behaves as if a branch has occurred to the first instruction to be traced. A full 32-bit branch address is broadcast to allow external reconstruction of the trace information. It is therefore recommended that in general the trace should not be disabled unless it will be off for a significant number of cycles. Otherwise, enabling and disabling the trace over just a few cycles will result in an increase in trace information passed through the trace port.

A simple way to turn the trace on and off is to set an address range inside which trace is enabled. Alternatively, a particular address or address range is used to turn on the trace port and another address or address range turns it off. The advantage of the second approach is that any subroutine calls that are outside of the function address range would also be included in the trace. It is anticipated that the sequencer is used for this purpose.

3.2.5 Data tracing

The trace port uses **ViewData** to control whether or not the information for a particular data access is broadcast in the trace stream. By reducing the amount of data trace that is broadcast the user can reduce the bandwidth required through the trace port and help prevent the on-chip FIFO from overflowing.

Typically the user would set a particular address range, which controls whether or not data accesses are broadcast. It is also possible to set start and stop conditions, so that data trace is enabled after a certain point in time and continues until it is later disabled.

Enabling and disabling data tracing based on the value of the data transferred is supported.

3.3 Trigger resources

To detect various interesting sequences of events that the processor performs, and thus trigger or enable tracing, the ETM has a configurable number of resources. The resource types consist of:

- address comparators
- data comparators
- memory map decoders
- EmbeddedICE module watchpoint comparators
- counters
- a three-state sequencer
- external inputs.

The numbers of resources are chosen at the time of synthesizing from HDL. This allows all unused hardware, and any redundancy due to unused resources, to be optimized away. The resource configuration of a particular ETM can be read via the JTAG interface.

The resources are classified according to whether they are:

- *Memory access resources*
- *Derived resources* on page 3-9
- *External inputs* on page 3-10.

The following sections describe the resource types, and *Resource identification* on page 3-10 describes the exact bit encodings for all types. This is used to allow a particular resource to be uniquely identified. In this manual resources are taken as being numbered from 1 to n.

3.3.1 Memory access resources

There are four types of memory access resource:

- single address comparators, used with or without data comparators
- address range comparators, used with or without data comparators
- EmbeddedICE module watchpoint comparators
- device specific memory map decoders.

Single address comparators

Address comparators compare either the instruction address or the data address against a user-programmed value. There are between zero and sixteen full address comparators. There must be an even number of full address comparators. Each address comparator may have a bit-masked data comparator. Each address comparator produces a *greater than or equal* (>=) and an *equal to* (=) output.

The memory access type and size (read/write, byte/word and so on) are also configured for each comparator.

Each comparator has a number of configuration bits to determine the match conditions. The available options are:

- instruction fetch
- instruction execute
- data load or store
- data load only
- data store only.

Instruction execute means the instruction at that address has reached the execute stage of the pipeline. Instructions that fail their condition codes are still considered to have executed. This is slightly different to the *instruction executed* (IE) pipeline status code, which considers condition code failure. *instruction not executed* (IN) is generated if the instruction reaches execution but fails its condition code test.

Note

When implementing an ETM this definition of IE means that more careful pipeline tracking is required because **nEXEC/INSTREXEC** from ARM7TDMI/ARM9TDMI cannot be used directly to determine this condition.

The address comparators are not bit-masked and this means that a single comparator cannot be used to generate a binary range (starts at an offset of 0 and ends at an offset of 2^n). If a binary range is required then it is necessary to use a pair of comparators referred to using the address range comparison, or to use the implementation-specific memory map decoders described below.

Alternatively, the ARM EmbeddedICE module allows full masked address comparisons to be carried out using a single EmbeddedICE comparator.

A memory access resource will only *match* for a single cycle, whether it be configured as a fetch or execute comparison. This is to allow such things as counters and sequencer transitions to occur cleanly, without the possibility of multiple counts or transitions due perhaps to memory wait states.

The 32-bit address from the ARM, which may or may not have Bits [1:0] masked, depending on whether these bits can be safely predicted, is compared with the address value.

Address range comparators

The full address comparators are arranged in pairs to form a *virtual* address range resource. The first comparator is programmed with the range start address and the second comparator is programmed with the range end address.

The resource matches if the address is then within the following range:

(address >= range start address) AND (address < range end address)

Features such as *out of range* are dealt with using Boolean operations available in the event logic described later.

Unpredictable behavior will occur if the two address comparators are not configured in the same way. For example, instruction fetch or execute, read or write and so on. The range comparison will not match if both of the comparator configurations do not also match.

An address range resource may *match* for a continuous number of cycles. The match will first occur at the point that the address is in the correct range. It will remain in this state until a new address outside the matching range is generated. Any access (read or write) outside the matching range will cause the range to go inactive.

When address comparators are used for comparisons on address ranges, a corresponding bit-masked data comparator can only be used in conjunction with the range start address comparator.

Data comparators

Each full address comparator is associated with a specific data comparator. A data comparator is only used to observe the data bus when load or store operations occur. Data comparisons on instruction fetches are not supported. Unpredictable behavior will result if a data comparison is enabled for an instruction fetch/execute resource.

A data comparison enable bit is provided for each data comparator implemented. The number of data comparators is customer defined. Between zero and eight address comparators may have associated data comparators. *Data comparators* on page 4-15 describes exactly how data comparators should be allocated to address comparators. As few or as many of the address comparators that are required may have associated data comparators, up to a maximum of eight.

A data comparator has both a value register and a mask register, so it is possible to only compare certain bits of the preprogrammed value against the data bus.

For information on data comparisons during aborts, see *Instruction fetch and data aborts* on page 3-21.

Address comparisons are always qualified by the data comparison. This also applies to address ranges.

EmbeddedICE comparators

The EmbeddedICE module watchpoint comparators can be used as additional trigger resources. The **RANGEOUT[1:0]** signals from the ARM processor are connected to the ETM.

Memory map decoder

Most system designs will contain an address decoder that divides the memory space into regions of RAM, ROM, peripherals and so on. The ETM can be customized with external logic for a particular application to allow low-cost decoding of address regions. As with the full address comparator resources, up to sixteen decodes are supported. An additional control register is provided to allow the user to statically configure the memory decode map to be used.

The interface to the external memory map decode (MMD) logic is implementation-specific. Refer to the appropriate Technical Reference Manual for details.

The memory map decoder behaves in a similar way to the address range comparators. In the case of the memory map decoder the full 32 bit address is used and any masking of addresses must be implemented externally.

The instruction and data addresses on which the decoder operates are registered. The memory map decode will become active when the address first matches, and will remain active until the comparison fails.

If precise memory comparisons are required, the behavior of the match must be checked to ensure it is only active for a single cycle. This ensures that the behavior is identical to the full address comparators described above.

The comparisons are likely to be simple bit-masked comparisons. This behavior is then similar to that of a typical memory decoder present in the ASIC memory system. The hardware required to implement this is also minimal.

The memory map decodes are only possible as *fetch* stage comparisons. No attempt is made to produce or select *execute* state versions. It is anticipated that these resources will primarily be used to decode the peripheral address map, and to subdivide the ARM code and data space. ViewData will be precise when based on memory map resources.

3.3.2 Derived resources

These are two types of derived resources that are not directly related to memory accesses. These are:

- 16-bit counters
- sequencer (state machine).

There are between zero and four counters and one or no sequencer.

Counters

Each counter is clocked by the system clock. The ARM wait signal is not taken into account. The counter decrements when its counter enable is active. This is controlled by:

- nothing, the counter decrements at the full system clock speed
- a count enable event.

The counters are 16-bit, so can count from 1 to 65535 events.

Each counter has a programmable preload register. When the register is first written, it is also copied into the counter. An event is defined which causes the counter to be reloaded from the register. The reload event takes priority over the count enable event.

When the counter reaches zero it remains at zero and the resource becomes active, and remains active until the counter is reloaded.

The current state of the counters can be accessed from read-only registers.

Sequencer states

A sequencer is provided and allows multiple level trigger sequences to be set up. The sequencer has three states. The state transitions are controlled with events. Each state has three possible next states (itself, and two others). Two events are therefore specified to control the state transitions. If neither event occurs or if both occur simultaneously, the sequencer will remain in the current state.

The current state of the sequencer can be accessed from a read-only register.

A full description of the sequencer and how it is programmed can be found in *Sequencer operation* on page 3-18.

3.3.3 External inputs

External inputs must be synchronous to the ETM clock. Within the ETM the external inputs are not related directly to memory accesses, so if they are used to enable/disable tracing in any way the effect will be imprecise.

Note

External input 16 (0x1111) is hard-wired to provide a permanently active resource.

3.3.4 Resource identification

Table 3-1 defines the available resource types and the bit encoding used to identify them.

Table 3-1 Resource identification encoding

Bit encoding	Range	Description
000	0 to 15	Address comparator. Produces = and >= outputs. The >= output is only used as part of the address range comparison.
001	0 to 7	Address range comparison. Uses pairs of address comparators.
010	0 and 1	EmbeddedICE module watchpoint comparators.
011	0 to 15	Memory map decodes.
100	0 to 3	Counter at zero.
101	0 to 2	Sequencer in a particular state.
110	0 to 3, 15	External inputs, hard wired input.
111		Reserved.

Note

Valid range encodings between 0 and 15 refer to resource IDs 1 to 16.

When referring to a particular resource being active this means that its output is a logical 1.

To identify a resource requires seven bits:

- three bits for the resource type
- four bits for the index.

The resource encoding is given in Table 3-2.

Table 3-2 Resource encoding

Bit numbers	Description
6:4	Resource type
3:0	Resource index

3.4 Example resource configuration

An example ETM resource configuration is shown in Figure 3-1, along with the various resource match signals that can be generated.

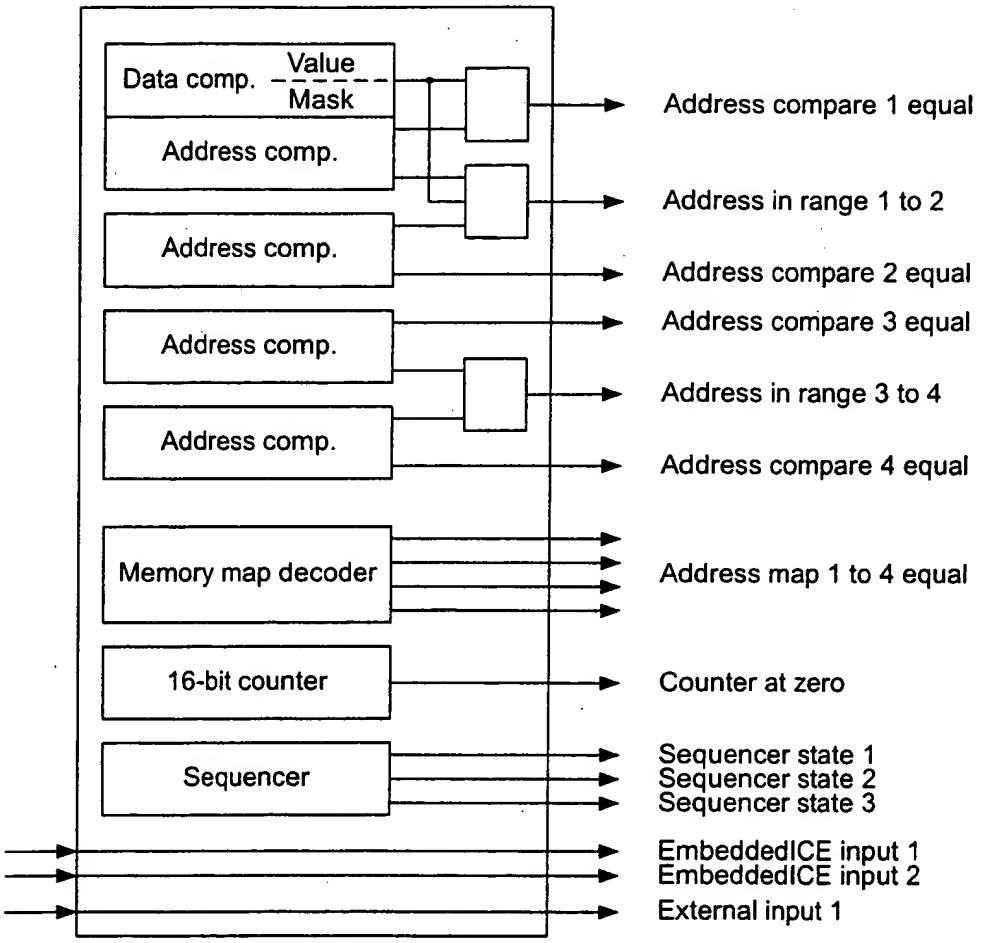

Figure 3-1 Example resource configuration

3.5 Events

Within this document, *event* refers to the basic Boolean combination of two resources.

Events control the basic transitions within the ETM, for example triggering and sequencer state changes. The combination of the two resource types allows many typical combinations such as *the 10th time the specified instruction is executed* to be easily expressed.

If we define A as the first resource match and B as the second match, an event is defined to be a function of A and B. The functions and the bit encodings are provided in Table 3-3.

Table 3-3 Boolean function encoding for events

Encoding	Function
000	A
001	Not (A)
010	A and B
011	Not (A) and B
100	Not (A) and Not (B)
101	A or B
110	Not (A) or B
111	Not (A) or Not (B)

A and B are identified with two seven bit fields. See *Resource identification* on page 3-10 for the exact resource encoding.

An event is encoded in three fields using a total of seventeen bits, shown in Table 3-4. The first two fields encode the two resources (see Table 3-1 and Table 3-2) and the third field encodes the Boolean operation to apply to them (see Table 3-3).

Table 3-4 Event encoding

Bit numbers	Description
16:14	Boolean function
13:7	Resource B
6:0	Resource A

Note

To permanently enable or disable an event, external input 16 should be specified, using either function A or Not (A).

3.6 Trace filtering

Tracing may be controlled in two ways:

- Tracing may be enabled and disabled dynamically based on events and resources (**TraceEnable** function).
- Data address and data data tracing may be enabled, either for regions of code or individual addresses (**ViewData** function). Again this is controlled with events and resources.

The trace filtering functions can be precise or imprecise, based on the resources used to control them.

3.6.1 Instruction tracing

The trace port uses the **TraceEnable** signal to turn tracing on and off. This signal has no effect if the trace port is disabled by the ETM control register.

TraceEnable is generated by:

- An enabling trace enable event. When the event is active tracing may occur.
- Either include or exclude address regions that are specified by the address range or memory map decode resources.

The operating mode (include or exclude) is statically defined for each trace run. Mixing include and exclude regions is not supported. Figure 3-2 on page 3-16 shows the structure of the **TraceEnable** signal.

An exclude region is useful for excluding library code or particular functions which are known to generate a lot of data.

An include region allows all code inside a simple range to be traced, for example the FIQ and IRQ handlers.

TraceEnable will only be precise if the resource that causes it to change is based on an execute stage memory access. This is because a branch destination address plus an address packet offset must be generated for the instruction fetched, ready for the instruction following it to be included in the trace.

PC tracing is imprecise in the following situations:

- if the resource that causes the function to become active is based on the data transferred by an instruction.
- if **TraceEnable** changes because the enabling event has changed, rather than an include/exclude region.

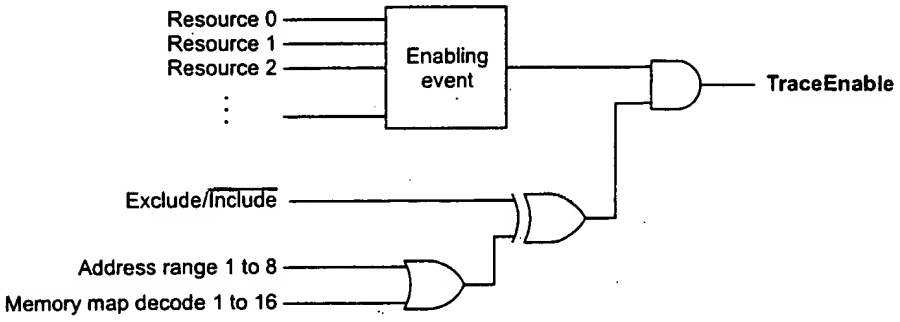

Figure 3-2 TraceEnable configuration

Note

If an access to a data variable is used to enable instruction or data tracing, that cycle, and sometimes the following cycle, will not be broadcast, thus losing visibility of the actual instruction and data that caused the enable event. If the trace is kept enabled for several cycles after this event the instruction may be inferred from the trace.

3.6.2 Data tracing

Data tracing is controlled in a similar way to **TraceEnable**. Control is provided in the following ways:

- An enabling event is used to control data tracing.
- Individual addresses, using the address comparator resources.
- Include and exclude address regions that are specified by the address range or memory map decode resources.

Data tracing is imprecise if **ViewData** changes because the enabling event has changed, rather than an include/exclude region.

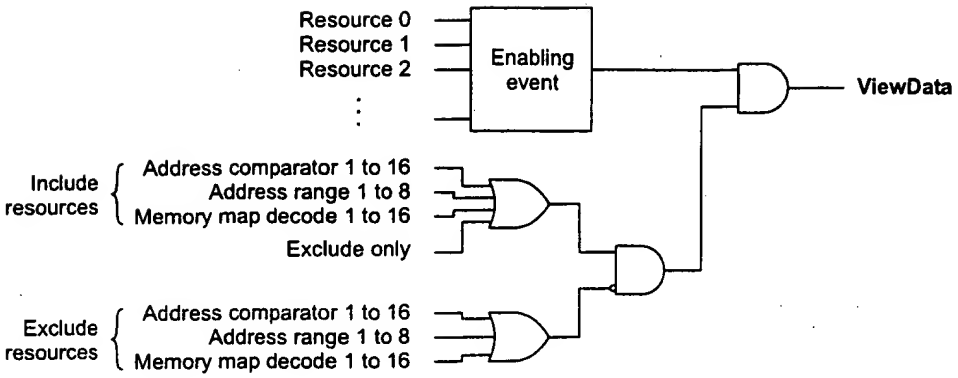

Figure 3-3 ViewData configuration

Exclude regions are provided to allow either large areas to be excluded, or to selectively exclude small regions or even single addresses from an include region. This might be from a memory map decoder. For example, an exclude region is defined as being not inside the specified address range.

That is when:

(address < range start address) AND (address >= range end address).

3.7 Sequencer operation

The sequencer is normally used to generate the trigger output. If the trigger is derived from a single event then the sequencer is not required, but when multiple stage trigger schemes are needed then the trigger event is normally based on a sequencer state.

The sequencer state diagram is shown in Figure 3-4. The sequencer has three states and for each state events are defined that control state transitions. On each cycle the sequencer can remain in the current state, move to the next state or return to a previous state.

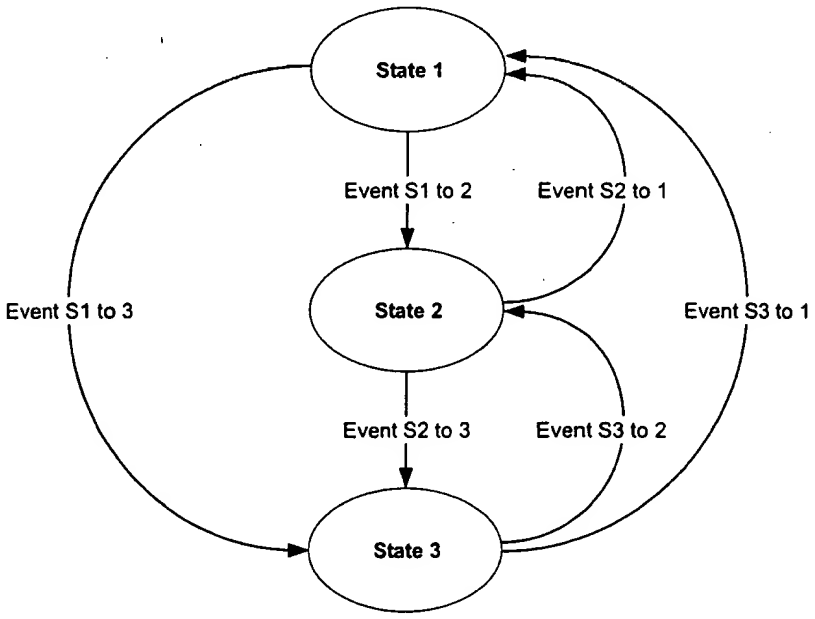

Figure 3-4 Sequencer state diagram

On reset, the sequencer goes to State 1.

The sequencer can change state on every clock cycle.

If two contradictory state transition events are active, no state transition occurs.

3.8 FIFO overflow

An optional ETM output called **FIFOFULL**, similar to **TraceEnable**, may be included in the ETM. Logic defines the address regions in which **FIFOFULL** may be asserted and a minimum empty byte count is provided to specify the point below which the FIFO is considered full.

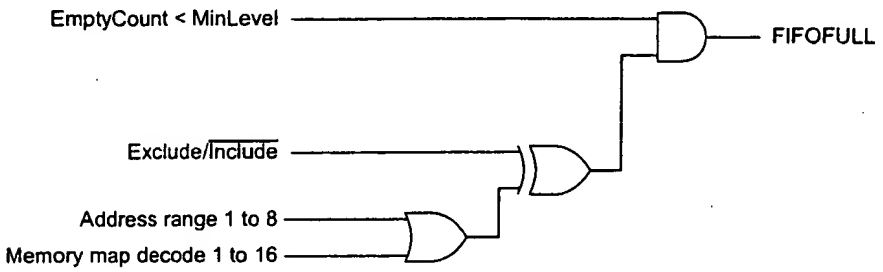

Figure 3-5 FIFOFULL generation

The **FIFOFULL** stall signal must be designed into the system since it is at that level that the stalling occurs. There is no support in an ARM core for an additional stall signal. The memory system normally asserts **nWAIT** to stall the core, generally based on address-based wait states or bus arbitration.

There are effects on interrupt latency that must be considered. If **FIFOFULL** is asserted, causing a load or store multiple (LDM/STM) instruction to be delayed, an IRQ or FIQ will not be taken until the delayed instruction completes. This means that the worst case interrupt latency may be affected.

FIFO overflow is independent of all resource matching, events and sequencer state changes. This will be undisturbed by a FIFO overflow.

3.9 External outputs

Some applications can make use of external outputs, possibly for the purposes of triggering a logic analyzer. Up to four external outputs are supported. Each output is controlled by an event, programmable in the normal way.

3.10 Instruction fetch and data aborts

In an environment that causes memory aborts, for example a demand-paged virtual memory system, the occurrence of aborts must be carefully managed. The programmer will not, in general, want to consider transfers that abort when counting accesses or generating external outputs, for example.

The trigger hardware must therefore take account of aborts when generating the following signals:

- **TraceEnable**
- **ViewData**
- **FIFOFULL**.

These will be asserted independent of both instruction and data aborts. For a load or store where the comparison is dependent on the data transferred, then if the data transfer is aborted the data comparison will be forced to match.

Note

ViewData may not be precise when dependent on a data comparison.

3.10.1 Events

Events drive the counters and control sequencer state transitions, for example. The following rules are applied for data aborts:

- A single data address compare does not ABORT. That is, it will match whether or not the access is aborted.
- A data address range compare does not consider ABORT. A range can only be used to count the number of cycles, not the number of individual accesses, in an address region.

The following rules are applied for instruction fetch (prefetch) aborts:

- A single instruction fetch stage address compare does not consider ABORT. This is because these resource matches are already imprecise.
- A single execute stage address compare considers ABORT. The address range compare will not match if the instruction is prefetch aborted.
- A single execute stage address compare on a load or store instruction does not consider any associated data aborts. The compare will match even if any or all of the data accesses are aborted.

When counting the execution of load or store instructions the occurence of data aborts and the subsequent retrying of instructions, will cause the instruction count to be larger than expected.

3.11 Usage examples

This section describes two examples which show how to set address ranges to cause specific event triggering

Example 3-1

To stop tracing on the 100th call of a function by another function:

1. A fetch address range is used to define when in the calling function.
2. Another address comparator is used to generate the counter enable event on each call.
3. When the counter reaches zero a trigger event can be generated.

Example 3-2

To trace all the code within a servo controller routine, but only to trigger when a value greater than x is written to the servo twice in succession:

1. An address comparator with associated data comparison generates a state transition event.
2. Another comparison detects a lower value, which again controls a state transition.
3. The trigger is generated when the sequencer is in a particular state. The data comparisons must be bit-wise.

3.12 Supported standard configurations

There are three standard configurations as described in Table 3-5. These configurations have been validated with the ARM Validation Suite and thoroughly tested with the ARM Software Toolkit.

Table 3-5 ETM configurations

Resource number/type	Small	Medium	Large
Pairs of address comparators	1	4	8
Data comparators	0	2	8
Memory map decoders	4	8	16
Counters	1	2	4
Sequencer present	No	Yes	Yes
External inputs	2	4	4
External outputs	0	1	4
FIFOFULL present	Yes	Yes	Yes
FIFO depth	9[a]/10[b]	18[a]/20[b]	45
Trace packet width[c]	4/8	4/8/16	4/8/16

a.ETM9
b.ETM7
c.Selectable using the ETM control register.

Chapter 4
Programmer's Model

This chapter describes the configuration registers that can be programmed to set up the trace and triggering facilities. It contains the following sections:

4.1 Overview of the ETM registers

When programming the various ETM registers it is important to enable all of the changes at the same time. For example, if the counter is reprogrammed, it could start to count, based on incorrect events, before the trigger condition has been correctly setup. In addition, the JTAG clock (**TCLK**) will often be asynchronous to the core clock.

The ETM programming bit in the ETM control register is used to disable all operations during programming. The following procedure must be followed:

1. Set the ETM programming bit.
2. Program all registers.
3. Clear the ETM programming bit.

The ETM registers are shown in Table 4-1. The ETM control register can be read and written. The ETM configuration code register, the counter value registers, and the current sequencer state register are read-only. All other registers are write-only.

Table 4-1 ETM register map

Register encoding	Function	Description
000 0000	ETM control	Controls the general operation of the ETM, such as whether tracing is enabled coprocessor data is traced and so on.
000 0001	ETM configuration code	Allows a debugger to read the number of each type of resource.
000 0010	Trigger event	Holds the controlling event.
000 0011	Memory map decode control	Eight-bit register, used to statically configure the memory map decoder.
000 0100	ETM status register[a]	Holds the pending overflow status bit.
000 0101	Reserved	
000 0110	Reserved	
000 0111	Reserved	
000 1000	**TraceEnable** event	Holds the enabling event.
000 1001	**TraceEnable** region	Holds the include and exclude regions.
000 1010	**FIFOFULL** region	Holds the include and exclude regions.
000 1011	**FIFOFULL** level	Holds the level below which the FIFO is considered full.
000 1100	**ViewData** event	Holds the enabling event.

Table 4-1 ETM register map (continued)

Register encoding	Function	Description
000 1101	**ViewData** control 1	Holds the include/exclude regions.
000 1110	**ViewData** control 2	Holds the include/exclude regions.
000 1111	**ViewData** control 3	Holds the include/exclude regions.
001 xxxx	Address comparator 1 to 16	Holds the address of the comparison.
010 xxxx	Address access type 1 to 16	Holds the type of access (for example instruction/data) and the size (for example byte or word).
011 xxxx	Data comparator values	Holds the data to be compared.
100 xxxx	Data comparator masks	Holds the mask for the data access.
101 00xx	Initial counter value 1 to 4	Holds the initial value of the counter.
101 01xx	Counter enable 1 to 4	Holds the counter clock enable control and event.
101 10xx	Counter reload 1 to 4	Holds the counter reload event.
101 11xx	Counter value 1 to 4	Holds the current counter value.
110 0xxx	Sequencer state and control	Holds the next state triggering events. Register address 7 is reserved. Register address 8 is used for reading the sequencer state.
110 10xx	External output 1 to 4	Holds the controlling events for each output.
110 11xx	Reserved	
111 0xxx	Implementation specific	Eight implementation specific registers.
111 1xxx	Reserved	

a.Only applies to protocol version 1 or later.

4.1.1 Reset behavior

The TAP reset (**nTRST**) input will cause the state of the ETM control register to be reset to the state described in Table 4-2 on page 4-7. In particular, the power-down and programming bits are set and the ETM selection bit is cleared. The trace port is disabled.

Setting the ETM programming bit will also cause the state of the ETM sequencer and counters to be reset, and cause tracing to be disabled.

Moving the the TAP state machine into Test-Logic Reset state resets only the TAP controller. No ETM registers are affected.

——— Note ———

To allow tracing on reset it is important that the JTAG reset signal (**nTRST**) is separate from the system reset.

On reset the status of registers or individual bits is unpredictable where not specified.

4.1.2 Implementation-specific registers

The addresses 111 0xxx in the register map are reserved for the future implementation of up to eight application-specific registers. These are intended for use by the ASIC designer to control resources outside the ETM, but within the ASIC, that may be used for trace functions.

——— Note ———

Implementors should be aware that trace debug tools may require application-specific modifications to support any added functionality.

4.1.3 Read-only registers

The current state of the counter and sequencer registers could change at any time, therefore if a read is performed asynchronously, the return values could be unreliable.

——— Note ———

Trace tool implementors should avoid setting the programming bit in the ETM control register prematurely, because this resets the counters and sequencer.

4.1.4 Event and resource programming

Events control the basic transitions within the ETM. An event is encoded with a function field (the event) and two resource fields (the resources), see Table 3-1 on page 3-10, Table 3-2 on page 3-11, Table 3-3 on page 3-13, and Table 3-4 on page 3-14 for details. A complete event and resource encoding is shown in Figure 4-1.

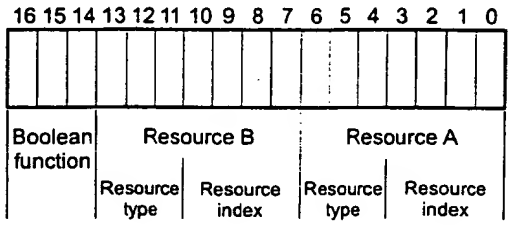

Figure 4-1 Event and resource encoding

Example 4-1 Event encoding

To encode a trigger event to occur when in address range 3 and when counter 2 reaches zero:

0x0100010010100001 would select the Boolean A AND Bfunction (010), together with address range comparator (001) for range 3 (0010) as resource B. Resource A is counter (100) number 2 (0001).

In addition, the trigger event register is programmed with the the trigger event that will occur when the Boolean expression becomes TRUE.

Example 4-2 Event encoding

To encode a TraceEnable event to occur when sequencer state 3 is reached:

00000000001010010 would select the Boolean A function. Resource A is defined as sequencer (101) state 3 (0010).

The TraceEnable event register is programmed with the TraceEnable event that will occur when the Boolean expression is TRUE.

Events will continue as long as the Boolean expressions that cause them remain TRUE, so that Example 4-1 and Example 4-2 both describe conditions that will cause single events.

4.2 Detailed register descriptions

This section describes the following registers in detail:

4.2.1 ETM control register

Table 4-2 describes the ETM control register.

Table 4-2 ETM control

Register bit	Control	Description
31:14	Reserved	Should be written as zero.
13	Half-rate clocking[a]	An output signal, controlled by this bit is provided to allow the trace clock to be divided by two when half-rate clocking is required. The external output is **CLKDIVTWOEN**. This bit should be set by the trace software tools to select the clocking mode required. On a TAP reset this bit is LOW.
12	Cycle-accurate tracing	In periods where tracing is enabled, this bit determines the level of **TRACEPKT[0]** for TD cycles. If **TraceEnable** is active and bit 12 is set HIGH, then **TRACEPKT[0]** is set HIGH to cause TD cycles to be captured. In all other cases, **TRACEPKT[0]** is set LOW and TD cycles are not captured. On a TAP reset this bit is LOW.
11	ETM port selection	An output, controlled by this bit is provided to allow the trace port pins to be shared with GPIO pins. The external pin is **ETMEN**. This bit should be set by the trace software tools to select between the available trace ports in a multi-processor ASIC. On a TAP reset this bit is LOW.
10	ETM programming	When HIGH the ETM is being programmed. All tracing is disabled and the sequencer, counters and so on are held in an inactive state. TD cycles will be visible on the trace port. On a TAP reset this bit is HIGH.
9	Debug request control	When HIGH the **DBGRQ** output is asserted until **DBGACK** is observed, when the trigger event occurs. This allows the ARM to be forced into Debug state. On a TAP reset this bit is LOW.
8	Branch broadcast	When HIGH all branch addresses are broadcast, even if the branch was due to a direct branch instruction. Enabling this bit allows real-time reconstruction of the processor address external to the device. On a TAP reset this bit is LOW.

Table 4-2 ETM control (continued)

Register bit	Control	Description
7	Stall processor	The **FIFOFULL** output can be used to stall the processor to prevent overflow. This signal is only enabled when the Stall processor bit is HIGH. When this bit is LOW the **FIFOFULL** output will remain LOW at all times and the FIFO will overflow if there are too many trace packets. On a TAP reset this bit is LOW.
6:4	Port size	The port size determines how many external pins are available to broadcast the trace information. This configuration determines how quickly the trace packets are extracted from the FIFO. 000 = 4-bit port 001 = 8-bit port 010 = 16-bit port 011 to 111 – reserved. On a TAP reset these bits are LOW.
3:2	Data access	00 = No data tracing 01 = Trace only the data portion of the access 10 = Trace only the address portion of the access 11 = Trace both the address and the data of the access. On a TAP reset these bits are LOW.
1	Monitor CPRT	When LOW the CPRTs are not traced. When HIGH the CPRTs are traced. On a TAP reset this bit is LOW.
0	ETM power down	A pin, controlled by this bit is provided to allow the ETM power to be controlled externally. The external pin is **PWRDOWN**. This bit should be cleared by the trace software tools at the beginning of a debug session. When HIGH the ETM should be powered down and disabled, and will then operate in a low power mode with all clocks stopped. On a TAP reset this bit is set HIGH.

a.Only applies to protocol version 1 or later.

Note

If a port width that is not supported by an ETM configuration is selected, the closest supported size is selected. The debug tools should read back the status register to confirm that the previous write was successful.

4.2.2 ETM status

Bit 0 of this read only register holds the pending overflow status bit. This register is only present for version 1 of the protocol or higher.

4.2.3 ETM configuration code

This register is read only, and allows the debugger to read the implementation-specific configuration of the ETM (see Table 4-3). Where the number of a particular resource is being specified, zero indicates that there are none of that resource type implemented.

Table 4-3 ETM configuration code

Bit numbers	Valid range	Description
31:28	0 to 15	ETM protocol version. The protocol version number can be used by the trace debug tools to identify the exact programmer's model for the ETM. The versions currently supported are 0000 and 0001.
27:24	-	Reserved. Should be zero.
23	-	When HIGH the **FIFOFULL** logic is present
22:20	0 to 4	Number of external outputs
19:17	0 to 4	Number of external inputs
16	-	When HIGH the sequencer is present
15:13	0 to 4	Number of counters
12:8	0 to 16	Number of memory map decoders
7:4	0 to 8	Number of data comparators
3:0	0 to 8	Number of pairs of address comparators

4.2.4 Trigger

A single trigger event register is required. This is shown in Table 4-4.

Table 4-4 Trigger event

Bit numbers	Description
16:0	Trigger event

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.5 Memory map decoder

The exact decode map is implementation-specific. At any one time there are only sixteen memory map resources available. Since many ASICs have a more complex memory map than this an 8-bit configuration register is provided to allow the memory map decode to be reconfigured (see Table 4-5). The size of this register is somewhat arbitrary as it simply allows the system designer to define multiple memory maps. It is likely that only one or two bits of this register are used.

For a Harvard ARM (separate instruction and data ports) it is up to the designer of the memory map decoder whether the same decode map applies to both the instruction and data address buses, or whether these are separately decoded. The latter is more sensible since, for example, instructions will never be fetched from the peripheral memory space. Alternatively, the configuration register could be used to control instruction/data address decoding.

Table 4-5 Memory map decode control

Bit numbers	Description
7:0	Memory map decode control

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.6 TraceEnable

Two registers are used. The first holds the enabling event (see Table 4-6).

Table 4-6 TraceEnable event

Bit numbers	Description
16:0	**TraceEnable** event

The second register holds the include/exclude resource control (see Table 4-7).

Table 4-7 TraceEnable region

Bit numbers	Description
24	Include/exclude control. When HIGH the specified resources indicate the regions to exclude tracing. When outside this region tracing may occur. When LOW the specified resources indicate the regions in which tracing may occur. When outside this region tracing is prevented.
23:8	Memory map decodes include/exclude control.
7:0	Address range comparator include/exclude control.

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.7 FIFO overflow

Two registers are required. The first controls the address regions in which **FIFOFULL** may be asserted, and the second the number of bytes below which the FIFO is considered full. Table 4-8 shows the **FIFOFULL** region register.

Table 4-8 FIFOFULL region

Bit numbers	Description
24	Include/exclude control. When HIGH the specified resources indicate the regions in which **FIFOFULL** cannot be asserted. When outside these regions **FIFOFULL** may be asserted. When LOW the specified resources indicate the regions in which **FIFOFULL** may be asserted. When outside these regions **FIFOFULL** cannot be asserted.
23:8	Memory map decodes include/exclude control.
7:0	Address range comparator include/exclude control.

Table 4-9 shows the **FIFOFULL** level register.

Table 4-9 FIFOFULL level

Bit numbers	Description
7:0	Byte count

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.8 ViewData

A single event plus three include/exclude control registers are required. Table 4-10 shows the **ViewData** event register.

Table 4-10 ViewData event

Bit numbers	Description
16:0	**TraceEnable** event

Table 4-11 shows the **ViewData** control 1 register, which is used as an include/exclude control for the address comparator resources.

Table 4-11 ViewData control 1

Bit numbers	Description
31:16	Address comparator exclude control
15:0	Address comparator include control

Table 4-12 shows the **ViewData** control 2 register, which is used as an include/exclude control for the memory map decoder resources.

Table 4-12 ViewData control 2

Bit numbers	Description
31:16	Memory map decode exclude control
15:0	Memory map decode include control

Table 4-13 shows the **ViewData** control 3 register, which is used as an include/exclude control for the address range comparator resources.

Table 4-13 ViewData control 3

Bit numbers	Description
16	Exclude only. When HIGH, **ViewData** is programmed only in an excluding mode. If none of the excluding resources match tracing may occur. When LOW **ViewData** operates in a mixed mode, and both include and exclude resources may be programmed.
15:0	Address range comparator exclude control.
7:0	Address range comparator include control.

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.9 Address comparators

Two registers are defined for each address comparator. The first holds the address to compare against (see Table 4-14).

Table 4-14 Address comparator value

Bit numbers	Description
31:0	Address value

The second holds a number of bits to define the type of access and to control whether bits 0 and 1 of the address are cleared or not (see Table 4-15).

Table 4-15 Address access type

Bit numbers	Description
5	Enable data comparison. When HIGH the address comparison is qualified by the corresponding data comparator.
4:3	Size mask: 00 – no mask 01 – clear A0 10 – reserved 11 – clear A0 and A1 If the size mask is set to 01 or 11, the associated address bits in the address comparator value register must be 0.
2:0	Access type: 000 – instruction fetch 001 – instruction execute 100 – data load or store 101 – data load 110 – data store 010, 011, 111 – reserved.

The access type value configures whether the address comparator is comparing instruction or data accesses, and for the latter whether only loads or stores will match.

The 32-bit address from the ARM may or may not have Bits [1:0] cleared by the size mask, depending on whether these bits can be safely predicted. The memory access size signals from the ARM are not used in the comparison. In general, comparisons on ARM instruction addresses should mask A0 and A1. Comparisons on Thumb instruction addresses should mask A0.

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.10 Data comparators

Two registers are defined for each data comparator. The first holds the 32-bit data value (see Table 4-16).

Table 4-16 Data comparator value

Bit numbers	Description
31:0	Data value

The second holds the 32-bit data mask.

Table 4-17 Data comparator mask

Bit numbers	Description
31:0	Data mask

Up to eight data comparators may be present.

A data comparison is permanently associated with a fixed-address comparator. If more than one data comparator is to be added, they are sequentially added to odd-numbered address comparators first (to provide maximum ranging capabilities), then to even-numbered comparators. The data comparator value register and data comparator mask register addresses directly correspond to equivalent address comparator register addresses.

For example, a system that uses five pairs of address comparators and six data comparators the address mapping would be as shown in Table 4-18.

Table 4-18 Example register address offsets

Address comparator number	Address comparator register offset	Data comparator?	Data comparator register offset
10	1001	No	-
9	1000	Yes	1000
8	0111	No	-
7	0110	Yes	0110
6	0101	No	-

Table 4-18 Example register address offsets (continued)

Address comparator number	Address comparator register offset	Data comparator?	Data comparator register offset
5	0100	Yes	0100
4	0011	No	-
3	0010	Yes	0010
2	0001	Yes	0001
1	0000	Yes	0000

When performing a data comparison the data mask register should be programmed appropriately to take account of both the size and the address offset of the transfer, to ensure that only valid byte lanes of the transfer are compared. (Refer to the appropriate memory interface chapter of the ARM core datasheet for details of how the lower address bits and the endianness affect how the ARM reads the data bus).

When a data mask bit is HIGH the corresponding bit in the value register is disregarded in the comparison.

The data comparator may only be used to observe data (that is, load/store accesses). If the address comparator is configured to match against instruction fetches the behavior will be unpredictable if the data comparator is also enabled.

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.11 Counter

Three registers are used to define the operation of the counter.

The first register is the initial count register which specifies the starting value of the counter. The counter is automatically loaded with this value when this register is programmed (see Table 4-19).

Table 4-19 Initial counter value

Bit numbers	Description
15:0	Initial count

Table 4-20 shows the counter enable register.

Table 4-20 Counter enable

Bit numbers	Description
17	Count enable source. When LOW the counter is continuously enabled. When HIGH the count enable event is used to enable the counter. It is recommended that bit 17 is always set HIGH and that the CountEnable event be used to control counter operation.
16:0	Count enable event.

Table 4-21 shows the counter reload register.

Table 4-21 Counter reload

Bit numbers	Description
16:0	Counter reload event

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.12 Sequencer

The sequencer is controlled by programming the events that are used to move between the different states (see Table 4-22).

Table 4-22 Sequencer state transition event

Bit numbers	Description
16:0	State transition event

A register for each event is provided to define the possible next states and the events that control the state transitions (see Table 4-23). The sequencer is reset to state one whenever any of the state transition event registers are programmed.

Table 4-23 Sequencer register allocation

Register number	Description
110 0000	State 1 -> State 2
110 0001	State 2 -> State 1
110 0010	State 2 -> State 3
110 0011	State 3 -> State 1
110 0100	State 3 -> State 2
110 0101	State 1 -> State 3
110 0110	Reserved
110 0111	Current sequencer state

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.2.13 External outputs

One event per output is required (see Table 4-24). These are held in separate registers.

Table 4-24 External output event

Bit numbers	Description
16:0	External output event

For details of event and resource encoding see *Event and resource programming* on page 4-5.

4.3 Programming and reading ETM registers

All registers in the ETM are programmed via a JTAG interface. The interface is an extension of the ARM TAP controller, and is assigned scan chain number 6.

The scan chain consists of a 40-bit shift register comprising:

- a 32-bit data field
- a 7-bit address field
- a read/write bit.

All registers in the ETM are configured via the JTAG interface. The general arrangement of the ETM JTAG registers is shown in Figure 4-2.

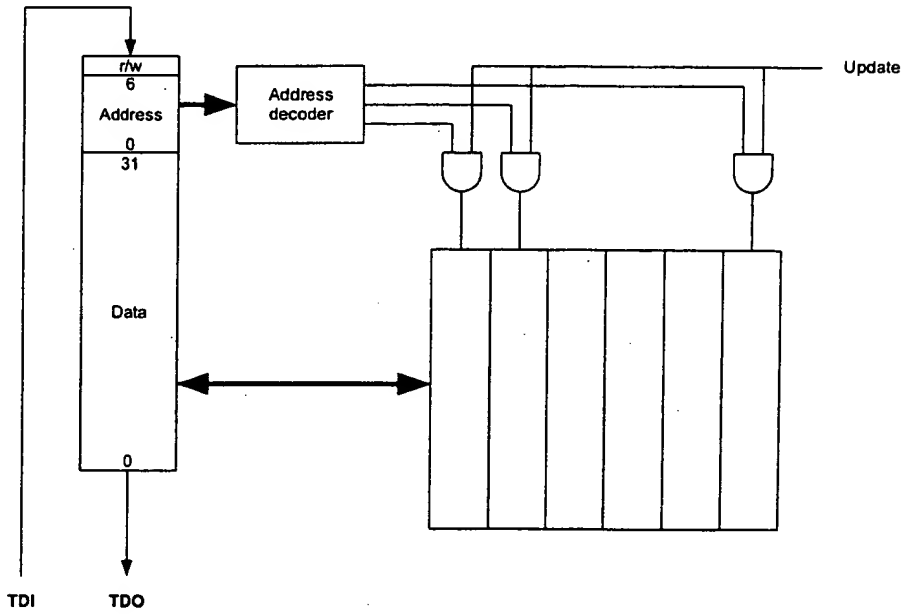

Figure 4-2 ETM JTAG structure

The data to be written is scanned into the 32-bit data field, the address of the register into the 7-bit address field and a 1 into the read/write bit.

A register is read by scanning its address into the address field and a 0 into the read/write bit. The 32-bit data field is ignored.

Note

A read or a write takes place when the TAP controller enters the UPDATE-DR state.

Chapter 5
Trace Port Physical Interface

This chapter describes the external pin interface, timing, and connector type required for the trace port on a target system. It contains the following sections:

5.1 Target system connector

A single AMP Mictor connector is specified. This is a high-density matched impedance connector. This connector has a number of important attributes:

- direct connection to a logic analyzer probe using a high-density adapter cable with termination (for example, HPE5346A from HP)
- matching impedance characteristics, allowing the same connector to be used up to 200MHz
- a large number of ground fingers to ensure good signal integrity
- inclusion of the run time control (JTAG) signals on the connector, allowing a single debug connection to the target.

Table 5-1 lists the AMP part numbers for the four possible connectors.

Table 5-1 Connector part numbers

AMP part number	Description
2-767004-2	Vertical, surface mount, board to board/cable connector.
767054-1	Vertical, surface mount, board to board/cable connector.
767061-1	Vertical, surface mount, board to board/cable connector.
767044-1	Right angle, straddle mount, board to board/cable connector.

The choice of connector used will depend on factors such as board thickness and cost. AMP connector distributors should be contacted for details of which connector best meets specific application requirements.

Table 5-2 provides the pinout for the target connector. All 38 pins are specified in this table. It is organized such that it can support:

- up to 16 trace data pins
- 3 pipeline status pins
- 1 trace sync pin
- 1 trace clock pin
- 1 external trigger pin
- 1 voltage reference pin
- 1 voltage supply pin
- 9 JTAG interface pins.

5.1.1 Single ETM target connector pinout

The pinout for a single processor ETM target connector is shown in Table 5-2.

Table 5-2 Single target connector pinout

Pin	Signal name	Pin	Signal name
38	**PIPESTAT[0]**	37	**TRACEPKT[8]**
36	**PIPESTAT[1]**	35	**TRACEPKT[9]**
34	**PIPESTAT[2]**	33	**TRACEPKT[10]**
32	**TRACESYNC**	31	**TRACEPKT[11]**
30	**TRACEPKT[0]**	29	**TRACEPKT[12]**
28	**TRACEPKT[1]**	27	**TRACEPKT[13]**
26	**TRACEPKT[2]**	25	**TRACEPKT[14]**
24	**TRACEPKT[3]**	23	**TRACEPKT[15]**
22	**TRACEPKT[4]**	21	**nTRST**
20	**TRACEPKT[5]**	19	**TDI**
18	**TRACEPKT[6]**	17	**TMS**
16	**TRACEPKT[7]**	15	**TCK**
14	VSupply	13	**RTCK**
12	VTRef	11	**TDO**
10	**EXTTRIG**	9	**nSRST**
8	**DBGACK**	7	**DBGRQ**
6	**TRACECLK**	5	**GND**
4	No connect	3	No connect
2	No connect	1	No connect

Note

Pins 1, 2, 3 and 4 *must* be true no-connects. For designs with less than sixteen trace data pins, pin 5 and any unused **TRACEPKT** pins *must* be connected to ground.

5.1.2 Dual ETM target connector pinout

The pinout for a dual processor ETM target connector is shown in Table 5-2. The **TRACECLK** signal is shared between the two trace ports that can be connected.

Table 5-3 Single target connector pinout

Pin	Signal name	Pin	Signal name
38	**PIPESTAT_A[0]**	37	**PIPESTAT_B[0]**
36	**PIPESTAT_A[1]**	35	**PIPESTAT_B[1]**
34	**PIPESTAT_A[2]**	33	**PIPESTAT_B[2]**
32	**TRACESYNC_A**	31	**TRACESYNC_B**
30	**TRACEPKT_A[0]**	29	**TRACEPKT_B[0]**
28	**TRACEPKT_A[1]**	27	**TRACEPKT_B[1]**
26	**TRACEPKT_A[2]**	25	**TRACEPKT_B[2]**
24	**TRACEPKT_A[3]**	23	**TRACEPKT_B[3]**
22	**TRACEPKT_A[4]**	21	**nTRST**
20	**TRACEPKT_A[5]**	19	**TDI**
18	**TRACEPKT_A[6]**	17	**TMS**
16	**TRACEPKT_A[7]**	15	**TCK**
14	VSupply	13	**RTCK**
12	VTRef	11	**TDO**
10	**EXTTRIG**	9	**nSRST**
8	**DBGACK**	7	**DBGRQ**
6	**TRACECLK_A**	5	**TRACECLK_B**
4	No connect	3	No connect
2	No connect	1	No connect

Note

Pins 1, 2, 3 and 4 *must* be true no-connects. For designs with less than sixteen trace data pins, unused **TRACEPKT** pins *must* be connected to ground.

5.1.3 Signal details

The signals on the target connector pins for both single and dual pinouts are described below.

TRACECLK, TRACESYNC, PIPESTAT, TRACEPKT

These are described in *Structure of the trace port* on page 2-3.

EXTTRIG

EXTTRIG is an optional signal. It is intended to be an input to one of the external inputs on the ETM. Depending on the design, ETM external triggers may not be available on the ASIC's external pins. In this case the **EXTTRIG** has no function. It is recommended that this pin is pulled to a defined state.

VTRef

The **VTRef** signal is intended to supply a logic-level reference voltage to allow debug equipment to adapt to the signalling levels of the target board. It does *not* supply operating current to the debug equipment. Target boards should supply a voltage that is nominally between 1V and 5V. With ±10% tolerance, this is minimum 0.9V, maximum 5.5V. The target board should provide a sufficiently low DC output impedance that the output voltage not change by more than 1% when supplying a nominal signal current (±0.4mA). Debug equipment that connects to this signal should interpret it as a signal rather than a power supply pin and not load it more heavily than a signal pin. The recommended maximum source or sink current is ±0.4mA.

VSupply

The **VSupply** signal allows the target board to supply operating current to debug equipment so that an additional power supply is not required. This may not be used by all debug equipment. The VDD power rail will typically drive the pin on the target board. Target board documentation should indicate the **VSupply** pin voltage and the current available. Target boards should supply a voltage that is nominally between 2V and 5V. With ±10% tolerance, this is minimum 1.8V. and maximum 5.5V. A target board that drives this pin should provide a minimum of 250mA, and 400mA is recommended. Debug equipment should indicate the required supply voltage range and the current consumption over that range. This enables the user to determine whether an external power supply is required to power the debug equipment. Target boards may have a limited amount of current available for external debug equipment, so a backup mechanism to power the debug equipment should be provided where **VSupply** is not connected or is insufficient. For some tools, this signal is unused.

nTRST

The **nTRST** signal is an open collector output from the run control unit to the **Reset** signal on the target JTAG port. This pin should be pulled high on the target to avoid unintentional resets when there is no connection.

——— **Note** ———

Board logic must ensure that there is a low pulse on the target ASIC's **nTRST** pin at power up.

TDI

TDI is the Test Data In signal from the run control unit to the target JTAG port. It is recommended that this pin is pulled to a defined state.

TMS

TMS is the Test Mode Select signal from the run control unit to the target JTAG port. This pin should be pulled up on the target so that the effect of any spurious **TCK**s when there is no connection is benign.

TCK

TCK is the Test Clock signal from the run control unit to the target JTAG port. It is recommended that this pin is pulled to a defined state.

RTCK

RTCK is the Return Test Clock signal from the target JTAG port to the run control unit. Some targets, such as ARM7TDMI-S or ASICs using TrackingICE technology, need to synchronize the JTAG port to internal clocks. To assist in meeting this requirement, a returned (and re-timed) **TCK** can be used to dynamically control the **TCK** rate. ARM's Multi-ICE run control product provides Adaptive Clock Timing, which waits for **TCK** changes to be echoed correctly on **RTCK** before making further changes.

TDO

This signal is the Test Data Out from the target JTAG port to the run control unit.

nSRST

This is an open collector output from the run control unit to the target system reset. This is also an input to the run control unit so that a reset initiated on the target may be reported to the debugger.

This pin should be pulled up on the target to avoid unintentional resets when there is no connection.

DBGRQ

The **DBGRQ** signal is used by the run control unit as a debug request signal to the target processor. It is recommended that this pin is pulled to a defined state. This signal is rarely implemented as a pin on the target ASIC.

This pin should be pulled down on the target to avoid unintentional debug requests when there is no run control unit connected.

DBGACK

The **DBGACK** signal is used by some run control units to detect entry or exit from debug state. This signal is rarely implemented as a pin on the target ASIC.

5.1.4 Connector orientation

The connector should be oriented on the target system as shown in Figure 5-1. This shows the view from above the PCB with the trace connector mounted near to the edge of the board. This allows the trace port analyzer to minimize the physical intrusiveness of the target interconnect, which may be PCB to PCB to ensure signal integrity.

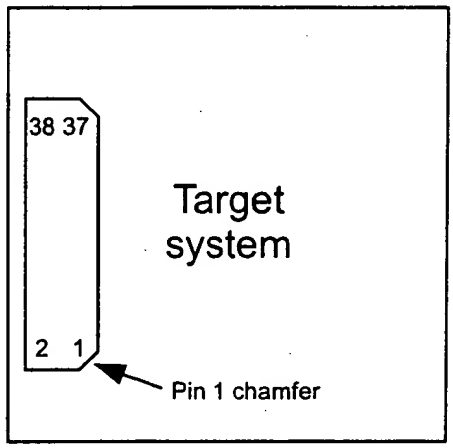

Figure 5-1 Recommended connector orientation

5.2 Timing specifications

There are no inherent restrictions on operating frequency, other than ASIC pad technology and trace port analyzer limitations. ASIC designers should provide a **TRACECLK** as symmetrical as possible, and with set up and hold times as large as possible. Trace port analyzer designers should conversely be able to support a **TRACECLK** as asymmetrical as possible, and require set up and hold times as short as possible. The following timing specifications are given as a guide for a Trace Port Analyzer which supports **TRACECLK** frequencies up to around 100MHz.

Note

Actual processor clock frequencies will vary according to application requirements and the silicon process technologies used. The maximum operating clock frequencies attained by ARM devices will increase over time as a result.

If the timing described here is adhered to it will be possible to use any ARM-approved TPA. Figure 5-2 depicts the timing for **TRACECLK**.

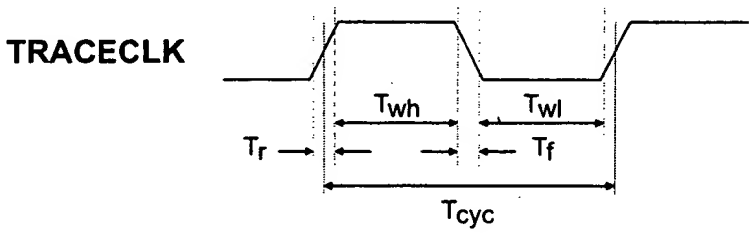

Figure 5-2 TRACECLK specification

Table 5-4 shows details of the **TRACECLK** parameters.

Table 5-4 TRACECLK parameters

Parameter	Minimum	Description
Tcyc	Frequency dependent	Clock period
Twl	2 ns	Low pulse width
Twh	2 ns	High pulse width

Figure 5-3 shows the setup and hold requirements of the trace data pins with respect to **TRACECLK**.

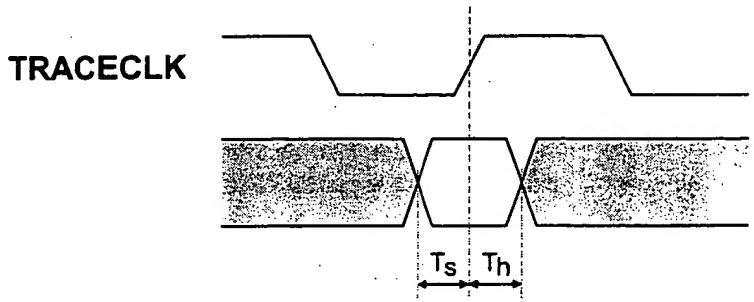

Figure 5-3 Trace data specification

Table 5-5 describes the timing for Figure 5-3.

Table 5-5 Trace port setup and hold requirements

Parameter	Min	Description
Ts	3 ns	Data setup
Th	2 ns	Data hold

5.2.1 Half-rate clocking mode

When half-rate clocking is used, the trace data signals will be sampled by the TPA on both the rising and falling edges of **TRACECLK**.

5.3 Signal level specifications

Debug equipment should be able to deal with a wide range of signal voltage levels. Typical ASIC operating voltages can range from 1V to 5V although 1.8V to 3.3V is typical.

Table 5-6 defines the allowable voltage levels on the trace port signals. A TPA must be able to operate with these voltage ranges. The two voltage ranges chosen reflect the two common supply voltage ranges currently in use.

Table 5-6 Voltage levels

Parameter	Description	2.5V		3.3V	
		MIN	MAX	MIN	MAX
Voh	High-level output voltage	2.0	-	2.4	-
Vol	Low-level output voltage	-	0.4	-	0.4

5.4 Other target requirements

It is important to keep the trace length differences as small as possible to minimize skew between signals. Crosstalk on the trace port should be kept to a minimum as it can cause erroneous trace results. Stubs on these traces can cause unpredictable responses, especially at high frequencies, so it is recommended that no stubs exist on the trace lines. If stubs are necessary, they should be made as small as possible.

The trace port lines (**TRACECLK**, **TRACESYNC**, **TRACEPKT** and **PIPESTAT**) should be series terminated as close as possible to the pins of the driving ASIC.

The maximum capacitance that will be presented by the trace connector, cabling and interfacing logic is 15pF.

For processor frequencies greater than 100MHz great care needs to be taken in the design of the input/output pads, chip package, PCB layout, and connections to the chosen Trace Port Analyzer. SPICE modeling is highly recommended.

5.5 JTAG control connector

At the time of writing (December 1999) all available JTAG controller products use a different connector to that specified in *Target system connector* on page 5-2. A target board must therefore either provide a second connector for the chosen JTAG controller or use an adapter board connected to the specified connector. If combined JTAG and trace analyzer products become available this requirement will not apply.

Index

The items in this index are listed in alphabetic order. The references given are to page numbers.

E

F

I

M

P

R

S

T

U

V